BRUCE ECKEL

THINKING IN JAVA

EDYCJA POLSKA

WYDANIE IV

http://helion.pl

Helion

Spis treści

Przedmowa

Początkowo Java była dla mnie „kolejnym językiem programowania" — jakim oczywiście pod wieloma względami jest.

Z biegiem czasu, oraz po głębszym poznaniu, zacząłem dostrzegać, iż podstawowe zamierzenie tego języka jest inne niż wszystkich, z którymi do tej pory się spotkałem.

Programowanie to radzenie sobie ze złożonością: złożoność problemu, który chcemy rozwiązać, potęguje złożoność maszyny, na której pracujemy. Właśnie z uwagi na złożoność większość projektów kończy się niepowodzeniem. Żaden z języków programowania, które poznałem, nie rozwinął się na tyle, aby *głównym* jego celem było przezwyciężanie złożoności samego programowania i późniejszej konserwacji gotowych programów[1]. Oczywiście wiele decyzji podczas projektowania języków zostało podjętych z myślą o ułatwieniach, ale okazywało się, iż zawsze istniały kwestie, które były uważane za niezbędne, i należy je dołączyć. Nieuchronnie te właśnie dodatki powodują, że programiści „przeginają", stosując taki język. Na przykład C++ musiał być zgodny wstecz z C (aby pozwolić programistom C na łatwą migrację) i do tego równie wydajny. Oba cele są bardzo ważne i znacznie przyczyniły się do sukcesu C++, lecz ujawniły również dodatkowe komplikacje, które uniemożliwiły ukończenie niektórych projektów (naturalnie można obwiniać programistów i menedżerów, ale jeżeli język może pomóc, wyłapując błędy, to dlaczego tego nie robi?). Kolejny przykład: Visual Basic (VB) był powiązany z BASIC-em, który nie został zaprojektowany jako rozszerzalny, stąd wszystkie dodatki nagromadzone w VB spowodowały powstanie naprawdę okropnej i niemożliwej do utrzymania składni. Perl jest zgodny wstecz z takimi narzędziami znanymi z systemu UNIX jak Awk, Sed, Grep i innymi, które miał zastąpić, a w rezultacie często jest oskarżany o generowanie „kodu tylko do zapisu" (co znaczy, że po kilku miesiącach nie można go zrozumieć). Z drugiej strony w C++, VB, Perlu i innych językach programowania, jak np. Smalltalk, projektanci skupili *część* wysiłku na kwestiach złożoności; w efekcie wszystkie te języki są skutecznie stosowane w odpowiednich dla nich dziedzinach.

To, co wywarło na mnie największe wrażenie, kiedy poznawałem Javę, to fakt, że wśród innych celów projektantów z firmy Sun znalazła się także redukcja złożoności *dla programisty*. To tak, jakby powiedzieć: „Nie dbamy o nic poza zmniejszeniem czasu i trudności tworzenia porządnego kodu". W początkowej fazie dążenie to kończyło się otrzymaniem

[1] Najbliżej tego celu jest według mnie Python — zobacz *www.Python.org*.

kodu, który niestety nie działał zbyt szybko (chociaż w przeszłości było wiele obietnic dotyczących zwiększenia szybkości działania programów), lecz zmniejszenie czasu pracy programisty było zdumiewające — potrzebował on teraz połowy lub nawet mniej czasu niż na napisanie równoważnego programu w C++. Przyczynia się to do dużych oszczędności, ale to nie wszystkie zalety Javy. Java idzie dalej, upraszczając wszystkie skomplikowane zadania, jak wielowątkowość i programowanie sieciowe, cechy języka lub biblioteki. Rozwiązuje także kilka naprawdę bardzo skomplikowanych problemów dotyczących programów działających na dowolnej platformie sprzętowej, dynamicznej zmiany kodu czy nawet bezpieczeństwa, które, na skali złożoności, można umieścić gdzieś pomiędzy „utrudnieniem" a „problemem nie do rozwiązania". Zatem pomimo problemów z wydajnością widać, iż możliwości Javy są olbrzymie, przez co może ona uczynić z nas znacznie wydajniejszych programistów.

Pod każdym względem — tworzenie oprogramowania, praca grupowa w celu tworzenia oprogramowania, budowa interfejsu użytkownika w celu komunikacji między programem a użytkownikiem, uruchamianie programów na różnych rodzajach maszyn i łatwe pisanie programów, które komunikują się poprzez Internet — Java zwiększa możliwości komunikacyjne *między ludźmi*.

Uważam, że wyniku rewolucji komunikacyjnej nie należy postrzegać jedynie przez pryzmat możliwości przesyłania ogromnych ilości bitów; dostrzeżmy prawdziwą rewolucję objawiającą się nowymi możliwościami porozumiewania się — indywidualnie, w grupach, ale również jako cała ludzkość. Chodzą słuchy, że następna rewolucja będzie polegała na powstaniu czegoś w rodzaju globalnego umysłu, tworzonego przez pewną grupę osób i odpowiednią liczbę połączeń między nimi. Java może — lub też nie — stać się narzędziem wspomagającym tę rewolucję. Już sama ta możliwość powoduje, że czuję, jakbym robił coś znaczącego, podejmując próbę nauczenia tego języka.

Java SE5 i SE6

Niniejsze wydanie książki czerpie całymi garściami z ulepszeń wprowadzonych do języka Java w jego wersji oznaczonej przez firmę Sun jako JDK 1.5, a potem przemianowanej na JDK5 albo J2SE5; wreszcie nazwa została ustalona jako Java SE5. Zmiany wprowadzone do Javy SE5 w znacznej mierze polegały na próbie jeszcze większego zbliżenia języka do programisty. Postaram się wykazać, że choć sukces nie jest całkowity, to faktycznie zmiany stanowią wielki krok we właściwym kierunku.

Jednym z celów tej edycji jest ujęcie ulepszeń Javy SE5 i SE6, ich przedstawienie i wkomponowanie w pierwotną treść książki. Oznacza to, że niniejsze wydanie jest ściśle ukierunkowane na specyfikę SE5 i SE6 i większości kodu dołączonego do książki nie da się skompilować za pomocą starszych wersji Javy; kompilator zgłosi błąd i zaniecha kompilacji. Myślę jednak, że korzyści płynące z udogodnień nowych wydań języka są warte tak radykalnej decyzji.

Tych, którzy w jakiś sposób są związani z poprzednimi wersjami Javy, odsyłam do poprzednich edycji niniejszej książki, udostępnionych w formie elektronicznej w witrynie *www.MindView.net*. Z różnych względów bieżące wydanie książki nie jest udostępniane w takiej formie — ze wskazanej witryny można za to pobrać wydania poprzednie.

Java SE6

Niniejsza książka to monumentalny, niezwykle czasochłonny projekt; w toku redakcji, zanim doszło do publikacji, pojawiła się wersja beta Javy SE6 (nazwa kodowa: *mustang*). Choć w Javie SE6 pojawiło się kilka pomniejszych zmian, które ulepszyłyby niektóre z przykładów prezentowanych w książce, zasadniczo pojawienie się wydania SE6 nie wpływa w znaczącym stopniu na aktualną jej treść; zmiany ograniczają się bowiem do usprawnień w bibliotekach i optymalizacji, a więc aspektów spoza głównego nurtu wykładu zawartego w książce.

Kod dołączony do wydawnictwa został pomyślnie przetestowany z próbnymi wersjami kompilatora Java SE6, nie spodziewam się więc zmian wpływających istotnie na treść książki. Gdyby takie zmiany pojawiły się w finalnej wersji Javy SE6, zostaną uwzględnione w kodzie źródłowym dołączonym do publikacji, który można pobrać ze strony *www.MindView.net*.

Okładka książki sugeruje, że jej treść dotyczy wydań SE5 i SE6, co należy rozumieć: „napisana pod kątem Javy SE5 z uwzględnieniem znaczących zmian języka wprowadzanych przez tę wersję, ale stosująca się również do przyszłej wersji SE6".

Przedmowa do wydania czwartego

Nowa edycja książki to satysfakcja z możliwości jej poprawienia, na bazie doświadczeń zdobytych od czasu publikacji poprzedniego wydania. Owe doświadczenia niejednokrotnie potwierdzają powiedzenie: „doświadczenie to coś, zyskiwanego przy porażce". Mam więc okazję do poprawienia tego, co dla Czytelników okazało się kłopotliwe albo żmudne. Jak zwykle, przygotowania do następnego wydania inspirują nowymi pomysłami, a trud aktualizacji jest z nawiązką rekompensowany wciąż intensywnym smakiem odkrycia i możliwością wyrażenia swoich pomysłów lepiej niż poprzednio.

Gdzieś tam kołacze się też świadomość, że trzeba to wydanie zrobić tak dobrze, żeby skłonić do jego zakupu posiadaczy wydań poprzednich. To silna presja do zwiększenia jakości, przepisania i reorganizacji wszystkiego, co tego wymaga — wszystko po to, by kolejne wydanie książki było nowym i wartościowym doświadczeniem dla tych, do których tę książkę kieruję.

Zmiany

Do poprzednich wydań książki dołączana była płyta CD-ROM; tym razem zrezygnowałem z niej. Zasadnicza treść owej płyty, w postaci multimedialnego seminarium *Thinking in C* (utworzonego dla MindView przez Chucka Allisona) jest teraz dostępna w postaci prezentacji Flash do pobrania z witryny MindView. Celem tamtego seminarium było przygotowanie Czytelników nieznających dostatecznie składni języka C do przyswojenia materiału prezentowanego w książce. Choć w dwóch rozdziałach zawarte jest szczegółowe omówienie składni języka, może być ono niewystarczające dla tych osób, które nie miały wcześniej styczności z C; *Thinking in C* jest właśnie dla nich — ma im pomóc osiągnąć odpowiedni poziom przygotowania do lektury.

Rozdział „Współbieżność" (który we wcześniejszych wydaniach nosił tytuł „Wielowątkowość") został przepisany z uwzględnieniem zmian wprowadzonych do bibliotek implementacji współbieżności w Javie SE5; wciąż zawiera jednak podstawowe wiadomości dotyczące pojęć związanych ze stosowaniem wątków. Bez tych podstaw zrozumienie bardziej zaawansowanych zagadnień wielowątkowości byłoby niezmiernie trudne. Spędziłem nad tym rozdziałem wiele miesięcy, nurzając się w krainie współbieżności; ostatecznie prezentuje on nie tylko podstawy, ale i wkracza na terytoria rezerwowane dla bardziej zaawansowanych.

Każde znaczące rozszerzenie języka, obecne w Javie SE5, jest omawiane w osobnym, nowym rozdziale książki. Omówienie zmian pomniejszych zostało włączone wprost do istniejącej bazy materiału. Z racji mojego osobistego zainteresowania wzorcami projektowymi w książce pojawiło się ich nieco więcej niż poprzednio.

Zasadniczej reorganizacji uległ układ materiału. Wynikła ona z obserwacji procesu nauczania oraz z pewnej zmiany mojego własnego postrzegania znaczenia słowa „rozdział". Poprzednio wydawało mi się, że rozdział konstytuuje materiał o dostatecznie dużej objętości, ale nauczając wzorców projektowych, przekonałem się, że słuchacze radzą sobie z materiałem najlepiej, kiedy zaraz po przedstawieniu wzorca przechodzimy do ćwiczeń z jego użyciem — efekt jest znakomity nawet wówczas, jeśli oznacza przerwanie ciągłości wywodu (przekonałem się przy okazji, że i mnie — jako wykładowcy — bardziej odpowiada taki rytm wykładu). W tym wydaniu starałem się więc łamać rozdziały wedle zagadnień, nie martwiąc się pozornym „niedomiarem" materiału w niektórych spośród nich. Sądzę, że to postęp.

Zdałem sobie także sprawę ze znaczenia testowania kodu. Bez wbudowanego systemu testującego zawierającego testy uruchamiane podczas każdego budowania systemu nie można mieć pewności, czy tworzony kod jest niezawodny. Aby ją uzyskać, na potrzeby niniejszej książki została stworzony specjalna infrastruktura testowania prezentująca i sprawdzająca poprawność wyników generowanych przez każdy program (dzieło to powstało w języku Python, a można je znaleźć w archiwum kodu źródłowego tej książki dostępnym w witrynie *www.MindView.net*). Testowanie jako takie zostało omówione w specjalnym dodatku, publikowanym pod adresem *http://MindView.net/Books/BetterJava*, przedstawiającym wszystkie zagadnienia traktowane aktualnie przeze mnie jako podstawowe umiejętności każdego programisty używającego Javy.

Poza tym przeanalizowałem każdy przykład przedstawiony w niniejszej książce, zadając sobie pytanie: „Dlaczego zrobiłem to właśnie w taki sposób?". W większości przypadków wprowadziłem pewne modyfikacje i poprawki, zarówno po to, by przykłady stały się bardziej spójne, jak również w celu przedstawienia najlepszych, w moim mniemaniu, praktyk programowania w Javie (oczywiście w ramach ograniczeń, jakie narzuca książka stanowiąca wprowadzenie do języka). Sporo istniejących przykładów zostało w znaczącym stopniu przeprojektowanych i zmodyfikowanych. Usunąłem także przykłady, które według mnie nie miały już sensu; dodałem też zupełnie nowe programy.

Czytelnicy nadesłali wiele wspaniałych komentarzy na temat trzech pierwszych wydań książki, co oczywiście było dla mnie niezwykle przyjemne. Niemniej jednak zawsze może się zdarzyć, że od czasu do czasu ktoś prześle uwagi krytyczne. Z jakiegoś powodu systematycznie pojawiały się uwagi, że: „Książka jest zbyt gruba". Osobiście sądzę, że jeśli jedynym zarzutem jest „zbyt duża liczba stron", nie jest to zbyt poważna krytyka

książki (znana jest uwaga cesarza Austrii dotycząca prac Mozarta, którym zarzucił „zbyt wiele nut!"; nie porównuję się bynajmniej z Mozartem). Poza tym mogę tylko zakładać, że uwagę taką nadesłała osoba nieświadoma ogromu języka Java, która jednocześnie nie widziała innych książek na ten temat. Niemniej jednak przygotowując niniejszą książkę, starałem się skrócić fragmenty zdezaktualizowane albo przynajmniej te, które nie mają kluczowego znaczenia. Ogólnie rzecz biorąc, starałem się wszystko przejrzeć, usunąć z tekstu tego wydania książki wszystkie niepotrzebne informacje, wprowadzić modyfikacje i poprawić wszystko, co tylko mogłem. Nie miałem żadnych oporów przed usuwaniem fragmentów tekstu, gdyż oryginalne materiały wciąż są na mojej witrynie WWW (*www.MindView.org*) jako ogólnie dostępne pierwsze, drugie i trzecie wydanie książki oraz dodatki.

Tych, którym ciągle przeszkadza objętość książki, najmocniej przepraszam. Możecie wierzyć bądź nie, bardzo się starałem, aby była mniejsza.

Projekt okładki

Okładka *Thinking in Java* jest inspirowana zaznaczającym się w zdobnictwie i architekturze nurtem rzemiosła artystycznego Arts & Crafts, mającym swój początek na przełomie zeszłych wieków, z apogeum w pierwszym dwudziestoleciu ubiegłego wieku. Nurt ten powstał w Anglii jako reakcja na produkcję taśmową i ogólnie Rewolucję Przemysłową z jednej strony oraz wysoce ozdobny styl wiktoriański z drugiej. Arts & Crafts to oszczędność formy, rękodzieło i szacunek dla indywidualnego wkładu rzemieślnika, co nie oznacza jednak rezygnacji ze stosowania nowoczesnych narzędzi. Dziś mamy do czynienia z echami tamtego nurtu: na przełomie wieków obserwujemy ewolucję od surowych początków rewolucji komputerowej (przesyłania bitów) w kierunku czegoś bardziej znaczącego, subtelniejszego; również dziś można mówić o rzemiośle programowania, które jest czymś więcej niż tylko produkowaniem kodu.

Tak samo postrzegam język Java: to próba wyniesienia programisty ponad surową mechanikę systemu operacyjnego z podniesieniem jego pracy do rangi rzemiosła nie tylko czysto użytkowego.

Zarówno autor tej książki, jak i projektant okładki (przyjaciele od dziecka) znaleźli w owym nurcie inspirację; obaj posiadają meble, lampy i inne elementy zdobnictwa i architektury wnętrz, które albo pochodzą wprost z tamtego okresu, albo są na jego dziełach wzorowane.

Inny motyw okładki kojarzy się z rodzajem gabloty, w której entomolodzy i zwykli zbieracze owadów mogliby prezentować okazy swojej kolekcji. Owe owady są obiektami, umieszczonymi w obiekcie gabloty. Sam obiekt gabloty został osadzony w obiekcie okładki; tak można zilustrować podstawową koncepcję agregacji w programowaniu obiektowym. Oczywiście programista skojarzy sobie zawartość gabloty z dręczącymi go „pluskwami"; na okładce widać owe „pluskwy": wyłapane, zidentyfikowane i (niestety) uśmiercone; można to odnieść do możliwości języka Java w zakresie wyszukiwania, prezentowania i eliminowania błędów (co zaliczam do silniejszych atrybutów tego języka).

Dla potrzeb niniejszego wydania przygotowałem akwarelę wykorzystaną jako tło dla okładki.

Podziękowania

W pierwszej kolejności chciałbym podziękować moim wspólnikom, którzy pracują ze mną, prowadząc kursy, konsultacje i tworząc projekty nauczania. Są to: Dave Barlett, Bill Venners, Chuck Allison, Jeremy Meyer i Jamie King. Jestem wdzięczny za Waszą cierpliwość w czasie, gdy usiłowałem opracować najlepszy model współpracy dla niezależnych gości, takich jak my.

Ostatnio, niewątpliwie za przyczyną Internetu, jestem kojarzony ze zdumiewająco liczną grupą osób, które wspomagają mnie w moich wysiłkach, zazwyczaj pracując w swych własnych domowych biurach. Swego czasu musiałbym całkiem sporo zapłacić za wynajęcie pomieszczeń biurowych zdolnych pomieścić wszystkie te osoby, jednak obecnie, dzięki Internetowi, poczcie i rozmowom telefonicznym, mogę korzystać z ich pomocy bez zbędnych kosztów. Wszyscy okazaliście się niezwykle przydatni w moich próbach „poprawienia kontaktów z innymi" i mam nadzieję, że wciąż będę mógł poprawiać swoją pracę, korzystając z wysiłków innych osób. Paula Steuer okazała się nieoceniona, jeśli chodzi o kontrolę moich dorywczych praktyk biznesowych i sprawienie, że nabrały one nieco sensu (dziękuję Ci, Paulo, za popędzanie mnie, gdy nie chce mi się czegoś robić). Jaśnie wielmożny pan Jonathan Wilcox poddał dokładnym badaniom strukturę mojej firmy, sprawdził każde potencjalne niebezpieczeństwo i przepchnął nas przez proces legalizacji naszych poczynań. Dziękuję za troskę i wytrwałość. Sharlynn Cobaugh (która odkryła Paulę) stała się ekspertem w dziedzinie przetwarzania dźwięku i w ogromnym stopniu przyczyniła się od powstania multimedialnych materiałów szkoleniowych oraz rozwiązania wielu innych problemów. Dziękuję Ci za wytrwałość w obliczu trudnych problemów komputerowych. Pracownicy firmy Amaio z Pragi pomogli mi w realizacji kilku projektów. Początkowym inspiratorem pracy przez Internet był Daniel Will-Harris i to dzięki niemu powstały wszystkie moje projekty graficzne.

Moim nieoficjalnym mentorem stał się na przestrzeni lat Gerald Weinberg, a to za sprawą jego konferencji i warsztatów, za które mu dziękuję.

Przy korekcie technicznej czwartego wydania nieoceniony okazał się Ervin Varga — choć nad jakością rozdziałów i przykładów pracowało również wielu innych. Ervin był głównym recenzentem technicznym książki, podjął się też zadania przepisania przewodnika po rozwiązaniach pod kątem nowego wydania. Znalazł trochę błędów i wprowadził szereg ulepszeń, które świetnie uzupełniały pierwotny tekst. Jego wnikliwość i pieczołowitość jest dla mnie fenomenem — to najlepszy korektor, jakiego znałem. Dziękuję, Ervin.

Kiedy trzeba było zastanowić się nad nowymi pomysłami, korzystałem ze swojego bloga prowadzonego w ramach witryny *www.Artima.com* pozostającej pod opieką Billa Vennersa. Czytelnicy tego forum, przesyłający swoje komentarze i uwagi, pomogli mi skrystalizować koncepcje. Korzystałem ze wskazówek Jamesa Watsona, Howarda Lovatta, Michaela Barkera i innych; szczególnie dziękuję tym, którzy pomogli mi w kwestii typów ogólnych.

Markowi Welshowi dziękuję za ciągłą pomoc i wsparcie.

Niezwykle pomocny okazał się ponownie Evan Cofsky, który arkana konfiguracji i opieki nad linuksowymi serwerami WWW wyssał najwyraźniej z mlekiem matki, dzięki czemu serwer witryny MindView jest wciąż dostępny i bezpieczny.

Specjalne podziękowania kieruję do mojej nowej przyjaciółki — chodzi o kawę, która podtrzymywała entuzjazm do projektu. Muszę przyznać, że Camp4 Coffee w Crested Buttle (w Kolorado) — gdzie zwykli przesiadywać słuchacze kursów MindView w czasie przerw — serwuje najlepszy na świecie wikt. Dziękuję Alowi Smithowi za powołanie tej placówki do życia i tchnięcie w nią tego świetnego ducha, który stanowi dziś nieodłączną część atmosfery Crested Buttle. Pozdrowienia dla wszystkich bywalców Camp4, ochoczo żłopiących tamtejsze napoje.

Dziękuję całej załodze wydawnictwa Prentice Hall; nieodmiennie otrzymuję od nich wszystko, czego potrzebuję, a bywam wybredny. Redaktorzy i redaktorki dokładają jednak starań, aby współpraca przebiegała bez zgrzytów.

Proces twórczy ujawnił nieocenioną przydatność niektórych narzędzi; za każdym razem, gdy z nich korzystam, jestem niezwykle wdzięczny ich twórcom. Biblioteka Cygwin (*http://www.cygwin.com*) rozwiązała niezliczone problemy, z którymi system Windows nie mógł (lub nie chciał) sobie poradzić; z każdym dniem stawałem się coraz bardziej przywiązany do niej (gdybym tylko ją miał 15 lat temu, kiedy mój mózg był ściśle związany z edytorem Gnu Emacs). IBM Eclipse (*http://www.eclipse.org*) jest naprawdę cudownym darem dla społeczności programistów, a ponieważ wciąż jest rozwijany, sądzę, że jeszcze dużo dobrego z niego wyniknie (od kiedy to IBM jest na fali? musiałem coś przegapić). Nowe ścieżki w narzędziach programistycznych pracowicie wytycza JetBrains IntelliJ Idea — również wiele na tym skorzystałem.

Przy pracach nad tą książką zacząłem korzystać z oprogramowania Enterprise Architect firmy Sparxsystems; szybko się okazało, że to mój ulubiony edytor języka UML. W wielu sytuacjach korzystałem z Jalopy — narzędzia formatującego kod autorstwa Marco Hunsickera (*www.triemax.com*); sam Marco pomógł mi skonfigurować program do moich specyficznych potrzeb. Od czasu do czasu korzystałem też z edytora programistycznego (i rozmaitych wtyczek do niego) JEdit Slavy Pestova — to zresztą świetny edytor dla słuchaczy moich seminariów.

I oczywiście, co powtarzam wystarczająco często i przy każdej sposobności, bezustannie rozwiązuję problemy, posługując się językiem Python (*www.Python.org*), wymyślonym przez mego znajomego Guido Van Rossuma oraz zabawnych geniuszy z PythonLabs z którymi spędziłem kilka wspaniałych dni wypełnionych programistycznym sprintem (Tim, oprawiłem w ramki pożyczoną od Ciebie mysz i oficjalnie nazwałem ją „TimMouse"). Ludzie, musicie jadać lunche w zdrowszym miejscu (dziękuję także całej społeczności osób używających Pythona, zdumiewającej grupie osób).

Mnóstwo osób przesyła mi poprawki i jestem im wdzięczny, ale w szczególności podziękowania niech przyjmą (za pierwsze wydanie): Kevin Raulerson (odnalazł tony błędów), Bob Resendes (po prostu niewiarygodny), John Pinto, Joe Dante, Joe Sharp (wszyscy trzej byli fantastyczni), David Combs (wiele gramatycznych i wyjaśniających poprawek), dr Robert Stephenson, John Cook, Franklin Chen, Zev Griner, David Karr, Leander A. Stroschein, Steve Clark, Charles A. Lee, Austin Maher, Dennis P. Roth, Roque Oliveira, Douglas Dunn, Dejan Ristic, Neil Galarneau, David B. Malkovsky, Steve Wilkinson

oraz całe rzesze innych osób. Prof. Marc Meurrens dołożył mnóstwo starań, by opublikować i udostępnić elektroniczną wersję pierwszej edycji książki w Europie.

Dziękuję wszystkim, którzy pomogli mi przepisać przykłady pod kątem biblioteki Swing (w drugim wydaniu książki) oraz pomagali mi w inny sposób. Są nimi: Jon Shvarts, Thomas Kirsch, Rahim Adatia, Rajesh Jain, Ravi Manthena, Banu Rajamani, Jens Brandt, Nitin Shivaram, Malcolm Davis oraz wszyscy ci, którzy udzielili mi pomocy.

W czwartym wydaniu korzystałem z uprzejmości Chrisa Grindstaffa, który pomagał mi w opracowaniu części poświęconej bibliotece SWT. Z kolei Sean Neville napisał pierwszą wersję części poświęconej Flex.

Spotkałem w swoim życiu mnóstwo uzdolnionych technicznie ludzi, którzy stali się moimi przyjaciółmi, a także wyróżniali się tym, że uprawiali jogę oraz praktykowali inne formy poprawy duchowej, według mnie, inspirujące i kształcące. Są to: Kraig Brockschmidt i Gen Kiyooka.

Nie jest dla mnie zaskoczeniem to, iż zrozumienie Delphi pomogło mi pojąć Javę, ponieważ wiele pojęć i założeń projektowych tych języków jest wspólnych. Moi przyjaciele od Delphi wsparli mnie w zgłębianiu tego cudownego środowiska programowania. Należą do nich: Marco Cantu (kolejny Włoch — czyżby przesiąknięcie łaciną dawało przepustkę do programowania?), Neil Rubenking (przed poznaniem komputerów był wegetarianinem, praktykował jogę i Zen) oraz oczywiście Zack Urlocker, stary kumpel, z którym podróżowałem po świecie. Wszyscy jesteśmy zaś dłużnikami Andersa Hejlsberga, który wciąż trudzi się z C# (który to język, jak się będzie można przekonać z książki, stanowił silne źródło inspiracji dla Javy SE5).

Spostrzeżenia i wsparcie mojego przyjaciela Richarda Hale'a Shawa (Kima również) były bardzo pomocne. Z Richardem spędziliśmy wiele miesięcy, prowadząc wspólnie seminaria i próbując opracować doskonałe przykłady dla uczestników.

Projekt książki, okładki i zdjęcie na okładce zostały wykonane przez mojego przyjaciela Daniela Will-Harrisa, znanego autora i projektanta (*www.Will-Harris.com*), który bawił się w szkole średniej stemplami czcionek, oczekując niecierpliwie na wynalezienie komputerów i możliwość składu komputerowego. Jednak to ja stworzyłem gotowe do składu strony, toteż błędy składu oryginalnego wydania książki są moje. Pisząc książkę, używałem Microsoft® Word XP dla Windows, a do stworzenia gotowych stron programu Adobe Acrobat; książka została wydrukowana bezpośrednio z plików PDF. W czasie powstawania ostatecznych wersji książki byłem akurat za granicą — pierwsze wydanie przesłałem z Capetown z Afryki Południowej, a drugie z Pragi — co nie byłoby możliwe bez zdobyczy wieku elektroniki. Wydania trzecie i czwarte ukończyłem w Crested Butte w Kolorado.

Szczególne podziękowania dla wszystkich moich nauczycieli i uczniów (którzy są równocześnie moimi nauczycielami).

Kiedy pracowałem nad tym wydaniem, moje kolana upodobała sobie kotka Molly — wypada i jej podziękować za ofiarowane mi ciepło.

Lista wspierających mnie przyjaciół (choć nie jest ona zamknięta) to: Patty Gast (fenomenalna masażystka), Andrew Binstock, Steve Sinofsky, JD Hildebrandt, Tom Keffer, Brian McElhinney, Brinkley Barr, Bill Gates z magazynu *Midnight Engineering*, Larry Constantine i Lucy Lockwood, Gene Wang, Dave Mayer, David Intersimone, Chris i Laura Strand, Almquistowie, Brad Jerbic, Marilyn Cvitanic, Mark Mabry i rodziny Robbinsów i Moelterów (oraz McMillansowie), Michael Wilk, Dave Stoner, Cranstonowie, Larry Fogg, Mike Sequeira, Gary Entsminger, Kevin i Sonda Donovanowie, Joe Lordi, Dave i Brenda Bartlettowie, Blake, Annette & Jade, Rentschlerowie, Sudeksowie, Dick, Patty i Lee Eckel, Lynn i Todd z rodzinami. No i oczywiście Mama i Tata.

Wprowadzenie

„On dał ludziom mowę, a mowa stworzyła myśl, która jest miarą Wszechświata"

— „Prometeusz rozpętany", Shelly

> *„Istoty ludzkie ... są pod wielkim wpływem języka stosowanego jako środek wyrażania w ich środowisku. Złudzeniem jest wyobrażać sobie, że jednostka dostosowuje się do rzeczywistości właściwie bez użycia języka – i że język jest tylko mało znaczącym środkiem rozwiązywania pewnych problemów komunikacji czy refleksji. Istotny jest fakt, że „świat realny" jest w znacznej mierze nieświadomie budowany na bazie językowych nawyków grupy."*
>
> *„The Status Of Linguistics As A Science", Edward Sapir, 1929*

Podobnie jak dowolny język naturalny Java zapewnia możliwość definiowania pojęć. Jednak, gdy rozwiązywany problem będzie się stawał coraz większy i coraz bardziej skomplikowany, to w sprzyjających okolicznościach okaże się, że ta forma przekazu będzie znacznie łatwiejsza i bardziej elastyczna niż formy alternatywne.

Nie można traktować Javy jako tylko zbioru cech — niektóre z nich osobno nie mają żadnego sensu. Możesz posłużyć się sumą tych części składowych tylko wtedy, kiedy myślisz o *projektowaniu*, a nie zwyczajnie o kodowaniu. Aby pojąć Javę w ten sposób, trzeba zrozumieć problemy w ujęciu języka oraz ogólnie pod kątem programowania. Ta książka omawia problemy programowania (skąd się biorą), jak również sposoby ich rozwiązywania z zastosowaniem Javy. Tak więc cechy, które opiszę w poszczególnych rozdziałach, oparte są na sposobie, w jaki postrzegamy konkretny rodzaj problemów pokonywanych z użyciem tego języka. Ten sposób, mam nadzieję, pozwoli osiągnąć Ci, drogi Czytelniku, stan, w którym świadomość Javy stanie się Twoim „naturalnym językiem".

Przez cały czas będę przyjmował, że chcesz zbudować w myślach pewien model, co pozwoli Ci dogłębnie zrozumieć język; jeżeli napotkasz problem, będziesz w stanie dopasować go do swojego modelu i znaleźć odpowiedź.

Warunki wstępne

Zakładam, że znasz pewne zagadnienia z dziedziny programowania: rozumiesz, że program jest zbiorem instrukcji, znasz pojęcie podprogramu, funkcji, makra, instrukcji sterującej typu „if" oraz konstrukcji pętli np. „while" itp. Mogłeś dowiedzieć się tego z wielu źródeł, np. programując w języku makropoleceń czy pracując z narzędziami podobnymi do Perla. Jeśli tylko programowałeś i poczułeś się swobodnie wykorzystując podstawowe aspekty programowania, będziesz również w stanie pracować z tą książką. Oczywiście będzie to łatwiejsze dla programistów C, a zwłaszcza programistów C++, lecz nie zniechęcaj się, jeśli nie znasz tych języków — o ile jesteś przygotowany na większy wysiłek. Odpowiednie przygotowanie można też zdobyć, studiując multimedialne seminarium „Thinking in C" publikowane w witrynie *www.MindView.net* i zawierające omówienie podstaw pomocnych w przyswajaniu tajników języka Java. W książce przedstawię natomiast zagadnienia programowania obiektowego (ang. *Object-Oriented Programming* lub w skrócie OOP) oraz podstawowe mechanizmy sterowania przebiegiem wykonania programu w języku Java.

Chociaż odwołania do cech języków C i C++ będą się pojawiać dosyć często, nie będą służyć jako komentarz, lecz pomagać programistom spojrzeć na Javę z perspektywy tych języków, z których przecież została wywiedziona. Spróbuję uczynić te odwołania prostymi i przedstawić je tak, aby wyjaśniały wszystko, z czym — jak myślę — programiści C i C++ mogli nie mieć do czynienia.

Nauka Javy

W tym czasie, w którym pojawiła się moja pierwsza książka *Using C++* (Osborne/McGraw-Hill, 1989), zacząłem uczyć tego języka. Uczenie języków programowania stało się moim fachem; od 1987 roku widywałem kiwające głowy, obojętne bądź zaskoczone twarze słuchaczy na całym świecie. Kiedy zacząłem prowadzić kursy w małych grupach, coś odkryłem. Otóż nawet ci, którzy się uśmiechali i przytakiwali, w wielu przypadkach byli zakłopotani. Zrozumiałem, przewodnicząc przez kilka lat ścieżce C++ (a później ścieżce Javy) na konferencji Software Development Conference, iż sam, podobnie jak inni przemawiający, mam zwyczaj prezentować zbyt wiele zagadnień w zbyt krótkim czasie. Toteż ostatecznie, biorąc pod uwagę zarówno zróżnicowanie poziomu odbiorców, jak i mój sposób prezentacji materiału, kończyłem z utratą części słuchaczy. Może to zbyt wiele, lecz z uwagi na to, że jestem jednym z przeciwników tradycyjnego prowadzenia wykładów (wydaje mi się, że wśród wielu osób brak zainteresowania jest wynikiem znudzenia), chciałem spróbować utrzymać wszystkich w tym samym tempie.

Przez jakiś czas tworzyłem kilka różnych prezentacji w dosyć krótkich odstępach czasu. Tak więc skończyłem na nauce poprzez eksperymenty i powtórzenia (technika, która sprawdza się również w przypadku projektowania programów w Javie). Ostatecznie opracowałem kurs z zastosowaniem tego wszystkiego, czego doświadczyłem jako wykładowca. Rozbija on proces nauki na oddzielne fragmenty, od łatwych do coraz trudniejszych, a na seminariach typu hands-on (idealne do nauki) po każdej z lekcji następują ćwiczenia.

Moja firma — MindView, Inc. — udostępnia go publicznie jako seminarium „Thinking in Java" przeznaczone do samodzielnego studiowania; jednocześnie jest on podstawowym kursem przygotowawczym do pozostałych, bardziej zaawansowanych seminariów. Szczegółowe informacje na ten temat można znaleźć w witrynie *www.MindView.net*. (Seminarium przygotowawcze dostępne jest także na CD-ROM-ie „Hands-On Java". Informacje na ten temat można znaleźć w firmowej witrynie WWW.)

Reakcje uczestników wykładów pozwalają mi na dokonywanie zmian w materiale lub skupienie się na coraz lepszych formach nauczania. Ale przecież książka nie jest opowieścią o wykładach — próbowałem upakować tak wiele informacji, ile tylko mogłem zmieścić na tych kilku ostatnich stronach, i ułożyć je tak, by wprowadzić Cię w kolejny temat. Książka jest przeznaczona przede wszystkim dla samotnych Czytelników, którzy zmagają się z nowym językiem programowania.

Cele

Podobnie jak moja poprzednia książka, pt. *Thinking in C++*, tak i ta została zaprojektowana z myślą o osobach poznających język. Kiedy myślę o rozdziale z książki, biorę pod uwagę to, jak stworzyć dobrą lekcję podczas seminarium. Słuchacze seminariów pomogli mi wytypować najtrudniejsze zagadnienia, wymagające szerszego komentarza. Tam, gdzie wiedziony ambicją wykładowcy starałem się zbyt szybko przekazać za dużą dawkę materiału, przekonałem się, że równoczesnej prezentacji wielu nowych elementów musi towarzyszyć wyczerpujące wyjaśnienie, a natłok zagadnień łatwo rozprasza i myli słuchaczy.

Dlatego każdy rozdział obejmuje teraz pojedyncze zagadnienie albo niewielką grupę zagadnień, bez uciekania się do koncepcji, które nie zostały jeszcze wyczerpująco omówione. Pozwala to na śledzenie wykładu i solidne przyswojenie materiału jeszcze przed przejściem do nowych kwestii.

Cele tej książki to:

1. Zaprezentować materiał w sposób prosty, tak aby dało się łatwo zgłębić każde zagadnienie przed dalszą lekturą. Starannie dobrana kolejność omawiania zagadnień ma każdorazowo zapewniać grunt do poznawania kolejnych. Oczywiście nie zawsze jest to możliwe — w takich sytuacjach pojawia się jedynie skrócona prezentacja zapowiadanych kwestii.
2. Stosować na tyle proste i krótkie przykłady, jak to tylko możliwe. Czasami powstrzymuje mnie to przed stawianiem czoła rzeczywistym problemom, ale odkryłem, że początkujący są zazwyczaj bardziej zadowoleni, rozumiejąc każdy detal niż będąc pod wrażeniem rozległości rozwiązanego problemu. Ponadto istnieje coś takiego jak ograniczenie ilości kodu, jaką da się przedstawić i przyswoić w warunkach wykładu. Bez wątpienia pojawią się opinie krytyczne, że używam „dziecinnych przykładów", ale godzę się na to w nadziei na uzyskanie użyteczności pedagogicznej.

3. Opisać to, co jest według mnie istotne, aby odbiorca zrozumiał język, a niekoniecznie wszystko, co sam wiem. Wierzę, że zachowuję hierarchię ważności wiadomości i że istnieją pewne fakty, których 95 procent programistów nigdy nie będzie potrzebować, a które po prostu wprawiają w zakłopotanie i świadczą o skomplikowaniu języka. Oto przykład pochodzący z C: jeżeli pamiętasz tabelę priorytetów dla operatorów (ja nigdy jej nie zapamiętałem), możesz napisać dosyć zmyślny kod. Jednak, jeśli się nad tym zastanowić, sprawia to kłopoty czytającemu i utrzymującemu taki kod. Toteż proponuję zapomnieć o priorytetach i używać nawiasów tam, gdzie tylko coś jest niezbyt jasne.
4. Opisać każdą część na tyle wyraźnie, aby czas wykładu oraz odstępy pomiędzy ćwiczeniami były niewielkie. Nie tylko pozwala to na utrzymanie odbiorców w większej aktywności i zaangażowaniu podczas seminariów, lecz również daje Czytelnikowi lepsze poczucie zrozumienia.
5. Dostarczyć solidne podstawy, tak by Czytelnik mógł pojąć koncepcję wystarczająco dobrze oraz przejść do zadań i książek trudniejszych.

Książka jako program nauczania

Pierwotne wydanie niniejszej książki powstało jako kompilacja materiałów z tygodniowego seminarium wykładanego jeszcze w czasach młodzieńczych języka Java; wtedy tydzień wystarczał. W miarę upływu czasu, nie bacząc na dojrzewanie i rozrost języka o nowe elementy i biblioteki, wciąż starałem się zachować tygodniowy rytm wykładu. W pewnym momencie jeden z klientów poprosił mnie, abym w organizowanym dla niego kursie ograniczył się do „samych podstaw", a kiedy próbowałem tego dokonać, odkryłem, że pompowanie takiej ilości materiału w ciągu tygodnia musiało być bolesne tak dla mnie, jak i dla moich słuchaczy. Java nie jest już prostym językiem, który można opanować w tydzień.

To doświadczenie i zyskana świadomość stały się motorem napędowym zmian; dziś seminarium o Javie zajmuje dwa tygodnie albo — w warunkach akademickich — dwa semestry. Część wprowadzająca kończy się na rozdziale „Obsługa błędów za pomocą wyjątków"; tak skomponowany kurs można jednak uzupełnić wprowadzeniem do JDBC, serwletów i JSP. Tak właśnie przedstawia się kompilacja kursu *Hands-On Java* (dostępnego na płycie CD-ROM). Reszta książki to materiał dla kursu stopnia średnio zaawansowanego, pokrywający się z seminarium (i płytą CD-ROM) *Intermediate Thinking in Java*. Obie płyty można nabyć za pośrednictwem wutryny *www.MindView.net*.

W sprawie dodatkowych materiałów dla wykładowców należy kontaktować się z wydawnictwem Prentice-Hall (*www.prenhallprofessional.com*).

Dokumentacja online

Język Java oraz dołączone biblioteki firmy Sun Microsystems (do ściągnięcia za darmo z witryny *java.sun.com*) wyposażone są w dokumentację elektroniczną, której można używać za pomocą przeglądarki internetowej. Podobnie jest z każdą implementacją Javy stworzoną przez inne firmy. Prawie wszystkie książki na temat Javy powielają zawartość tej dokumentacji. Tak więc albo już ją posiadasz, albo możesz ją zdobyć. Jeżeli nie będzie takiej konieczności, nie będę powtarzał tego, co jest w dokumentacji, gdyż zazwyczaj znacznie szybsze jest odnalezienie opisu klasy w przeglądarce niż poszukiwanie jej w książce (poza tym dokumentacja online jest z pewnością bardziej aktualna). Zamieszczę zatem wyłącznie informacje, że należy zajrzeć do „dokumentacji JDK". W książce znajdziesz natomiast dodatkowy opis klas, ale tylko wtedy, gdy uzupełnienie dokumentacji będzie niezbędne do wyjaśnienia konkretnego przykładu.

Ćwiczenia

Zauważyłem, że proste ćwiczenia bardzo ułatwiają pełne zrozumienie tematu przez słuchacza uczestniczącego w seminarium, toteż na końcu każdego z rozdziałów zawsze znajdziesz listę ćwiczeń.

Większość zadań została obmyślona tak, aby były na tyle proste, by można je wykonać w rozsądnym czasie w warunkach szkolnych, podczas gdy prowadzący mógłby obserwować i upewnić się, czy wszyscy zrozumieli materiał. Niektóre stanowią większe wyzwanie, lecz nie są to poważne problemy.

Rozwiązania do wybranych zadań można odnaleźć w dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za drobną opłatą na stronie *www.MindView.net*.

Przygotowanie do lektury

Kolejną niespodzianką dla Czytelników niniejszego wydania jest darmowy multimedialny kurs do pobrania z witryny *www.MindView.net*. Mowa o seminarium *Thinking in C*, które stanowi wprowadzenie do składni języka C, stosowanych w nim operatorów i funkcji — na których przecież bazuje składnia języka Java. W poprzednich wydaniach materiał ten rozprowadzany był pod nazwą *Foundations for Java* na płycie CD-ROM dołączanej do książki. Teraz można pobrać go za darmo.

Swego czasu zachęcałem Chucka Allisona do utworzenia *Thinking in C* jako samodzielnego produktu, ale potem zdecydowałem się na dołączenie go do drugiego wydania *Thinking in C++* oraz drugiego i trzeciego wydania *Thinking in Java* — chodziło mi o ujednolicenie przygotowania moich słuchaczy, którzy dzięki dostępności tego kursu mogli wyrównać swoje wiadomości odnośnie podstawowej składni języka C. Zwykle słuchacze prezentują postawę: „Jestem bystry i nie będę zniżał się do nauki C; dam sobie radę z C++ (Javą)", ale po przybyciu na seminarium przekonują się, że wymóg znajomości podstawowej składni języka C nie jest czczym wymysłem.

Technologie się zmieniają i uznaliśmy, że sensowniej jest udostępniać seminarium w postaci dającej się pobrać na komputer lokalny prezentacji Flash, rezygnując z rozprowadzania płyt CD-ROM. Powszechna dostępność kursu na stronach WWW daje też lepszą gwarancję odpowiedniego przygotowania słuchaczy wykładów.

Kurs *Thinking in C* niejako poszerza grono odbiorców niniejszej książki. Choć rozdziały „Operatory" i „Sterowanie przebiegiem wykonania" zawierają omówienie najważniejszych elementów języka Java wywodzących się właśnie z C, seminarium online stanowi łagodniejsze wprowadzenie, bo stawia odbiorcom mniejsze wymagania co do posiadanej umiejętności programowania.

Kody źródłowe

Wszystkie kody dołączone do książki są chronionym prawem autorskim oprogramowaniem darmowym, rozpowszechnianym jako pojedynczy pakiet na stronie *www.MindView.com*. Aby upewnić się, że dysponujesz najaktualniejszą wersją, pamiętaj, że jest to oficjalna witryna dystrybucji kodu oraz książki w postaci elektronicznej. Możesz swobodnie rozpowszechniać ten kod na zajęciach, wykładach czy szkoleniach.

Zasadniczym celem ochrony praw autorskich jest zapewnienie, by kody źródłowe były właściwie przytaczane oraz by powstrzymać Cię przed ich publikowaniem w postaci wydrukowanej bez odpowiedniej zgody (dopóki kody są cytowane wraz ze źródłem pochodzenia, użycie przykładów z książki w większości środków przekazu nie stanowi problemu).

Przetłumaczone na język polski kody znajdziesz pod adresem *ftp://ftp.helion.pl/przyklady/thija4.zip*. W każdym pliku źródłowym znajdziesz odwołanie do poniższej informacji o prawach autorskich:

```
//:! :CopyRight.txt
Copyright ©2000 Bruce Eckel
Plik kodu źródłowego z polskiej wersji czwartego wydania
książki "Thinking in Java". Wszystkie prawa zastrzeżone
POZA pozwoleniami przedstawionymi poniżej:
Plik ten może być wykorzystany nieodpłatnie
do celów własnych (osobistych lub komercyjnych),
włącznie z możliwością zmian i rozpowszechniania
tylko w formie wykonywalnej. Kodu można użyć na
zajęciach, jeżeli tylko książka "Thinking in Java"
zostanie wymieniona jako jego źródło.
Poza zajęciami szkolnymi kod nie może być kopiowany
i rozpowszechniany: wyłącznym miejscem dystrybucji
jest http://www.MindView.com (i oficjalne
serwery lustrzane), gdzie jest udostępniany
nieodpłatnie. Nie można usuwać tej informacji
o prawach autorskich ani też rozpowszechniać
zmodyfikowanej wersji kodu z tego pakietu.
Nie można używać pliku w postaci wydrukowanej
bez specjalnej zgody autora. Bruce Eckel nie bierze
odpowiedzialności za stosowanie tego oprogramowania
w jakimkolwiek celu. Kod jest dostarczony w niniejszej postaci
bez żadnej gwarancji.
```

```
Całkowite ryzyko związane z jakością i wydajnością oprogramowania
ponosi użytkownik. Bruce Eckel oraz wydawnictwo nie ponoszą
żadnej odpowiedzialności za zniszczenia spowodowane
przez użytkownika lub kogokolwiek innego w wyniku
stosowania lub rozpowszechniania oprogramowania.
W żadnym razie autor i wydawca nie podlegają
odpowiedzialności za jakiekolwiek straty finansowe
lub utratę danych czy też pośrednie lub bezpośrednie, celowe
lub przypadkowe zniszczenia będące efektem stosowania
lub niemożności zastosowania oprogramowania, nawet
gdy zostaną poinformowani o możliwości takowych zniszczeń.
Czy w oprogramowaniu powinno się poprawiać usterki?
Należy wziąć pod uwagę ogrom niezbędnych uzupełnień,
napraw i korekt. Jeśli uważasz, że znalazłeś błąd, to
bardzo proszę o jego poprawienie w oryginalnym tekście
za pomocą formularza ze strony www.MindView.com
(proszę również o użycie tego samego formularza
w przypadku znalezienia w książce błędów
niezwiązanych z kodem).
///:~
```

Możesz używać kodów we własnych projektach oraz podczas zajęć (włączając w to własne materiały prezentacyjne), jeżeli tylko zachowasz informację o prawach autorskich znajdującą się w poszczególnych plikach źródłowych.

Konwencje zapisu

W treści książki wyrazy będące identyfikatorami (funkcje, zmienne i nazwy klas) są pisane czcionką pogrubioną o stałej szerokości znaku. Podobnie pisanych jest wiele innych słów kluczowych poza tymi, które są na tyle często stosowane, że dodatkowe oznaczenie może stać się nudne, jak np. „class".

Stosuję również specjalny styl w przypadku przykładowych kodów. Styl ten odpowiada konwencjom stosowanym przez firmę Sun we wszystkich kodach źródłowych, które można spotkać w jej witrynie (*java.sun.com/docs/codeconv/index.html*) i, zdaje się, jest obsługiwany przez większość środowisk programowania Javy. Jeśli zapoznałeś się z moimi innymi pracami, musiałeś zauważyć, że stosowana przeze mnie konwencja jest bardzo podobna — co mnie cieszy, chociaż nic mi to nie daje. Temat konwencji zapisu jest dobry na długie debaty, zatem oznajmiam, że nie próbuję narzucać poprawnego zapisu poprzez zamieszczone przykłady; mam swoje powody, aby stosować taki zapis, jaki stosuję. Ponieważ Java jest językiem programowania zezwalającym na dowolną formę zapisu, toteż możesz stosować taki styl, z jakim Ci jest najwygodniej. Zakłopotanym utrzymaniem jednolitego stylu kodowania polecam program w rodzaju Jalopy (*www.triemax.com*), którego osobiście używałem do formatowania kodu.

Pliki kodu źródłowego przedrukowane w książce zostały przetestowane zautomatyzowanym systemem kompilacji, powinny więc dawać się kompilować bez błędów.

Niniejsze wydanie jest przeznaczone dla Javy SE5 (SE6). Wszystkich zainteresowanych poznaniem poprzednich wersji języka (nieomawianych w tym wydaniu) odsyłam do wydań poprzednich (pierwszego, drugiego i trzeciego), dostępnych nicodpłatnie w formie elektronicznej w witrynie *www.MindView.net*.

Błędy

Niezależnie od metod, jakich użyje autor do wykrywania błędów, to i tak jakieś zawsze się wkradną do tekstu i często rzucają się w oczy wnikliwym Czytelnikom. Jeśli znajdziesz coś, co uważasz za błąd, skorzystaj, proszę, z systemu BackTalk i prześlij informację o błędzie oraz sugerowane korekty. Doceniam i dziękuję za pomoc.

Rozdział 1.

Wprowadzenie w świat obiektów

„Dzielimy otaczający nas świat i kategoryzujemy go, przypisując po drodze tym kategoriom znaczenia — ponieważ jesteśmy częściami społeczeństwa bazującego na mowie, którego sens jest w mowie zakodowany... Nie moglibyśmy porozumiewać się, nie stając się częścią tej społeczności i nie korzystając z tego systemu klasyfikacji."

— Benjamin Lee Whorf (1897 – 1941)

Genezą rewolucji komputerowej była maszyna — dlatego genezą języków programowania było „udawanie" maszyny.

Komputery jednak bardziej niż maszynami są „wzmacniaczami umysłu" (lub „rowerami dla umysłu" — jak mawia Steve Jobs), a także nowym środkiem ekspresji. W rezultacie w coraz mniejszym stopniu przypominają maszyny, a w coraz większym fragmenty naszego umysłu, a zarazem klasyczne dziedziny sztuki, takie jak: literatura, malarstwo, rzeźba, animacja i film. Programowanie zorientowane obiektowo jest częścią ruchu chcącego uczynić z komputera kolejny środek ekspresji.

W tym rozdziale przedstawię podstawowe idee programowania zorientowanego obiektowo (ang. *Object Oriented Programming*, OOP). Dokonam także przeglądu obiektowych metod tworzenia oprogramowania. Podobnie jak reszta książki, rozdział ten zakłada pewien poziom doświadczenia Czytelnika w programowaniu proceduralnym, choć niekoniecznie w języku C. Kto nie czuje się mocny w tej dziedzinie, powinien przejść multimedialny kurs *Thinking in C* dostępny do pobrania na stronach witryny *www.MindView.net*.

Treść rozdziału stanowi podstawę, a jednocześnie materiał uzupełniający resztę książki. Wiele osób nie chce zagłębiać się w programowanie obiektowe bez uprzedniego zrozumienia jego ogólnych koncepcji — dlatego opiszę tu wiele z nich, tworząc w ten sposób solidny obraz całej techniki. Z drugiej jednak strony są tacy, którzy mają problemy ze zrozumieniem ogólnych idei przed poznaniem „mechaniki" — ludzie ci mogą poczuć się zagubieni i zakłopotani bez jakiegoś fragmentu kodu jako punktu zaczepienia. Jeżeli należysz do tej drugiej grupy i nie możesz się już doczekać omawiania specyfiki języka, oczywiście

możesz opuścić ten rozdział — na tym etapie nie uniemożliwi to pisania programów i nauki języka. Powinieneś jednakże powrócić do niego później, aby zrozumieć, dlaczego obiekty są ważne i jak przy ich użyciu projektować oprogramowanie.

Postępująca abstrakcja

Każdy język programowania dostarcza pewnej abstrakcji. Można próbować bronić tezy, że złożoność problemów, jakie jesteśmy w stanie rozwiązać, jest bezpośrednio zależna od rodzaju i jakości używanej abstrakcji. Przez „rodzaj" abstrakcji rozumiem odpowiedź na pytanie: „Czym jest to, czego abstrakcji dokonujemy?". Język asembler jest niewielką abstrakcją maszyny. Wiele tzw. imperatywnych języków, które stworzono później (takich jak Fortran, BASIC i C), stanowiło niewielką abstrakcję asemblera. Były w porównaniu z nim znacznym usprawnieniem, jednak pierwotna abstrakcja pozostała niezmieniona — wciąż wymagane było myślenie w kategoriach struktury komputera, a nie struktury rozwiązywanego problemu. Programista musiał dokonać powiązania pomiędzy modelem maszyny (w „przestrzeni rozwiązania", czyli miejscu, w którym modelujemy problem, takim jak komputer) a modelem samego rozwiązywanego problemu (w „przestrzeni problemu", czyli miejscu, w którym problem istnieje, np. w działalności firmy). Wysiłek wymagany przy konstrukcji takiego odwzorowania w połączeniu z faktem, iż jest ono czymś zewnętrznym w stosunku do języka programowania, powoduje, że programy są trudne do napisania i kosztowne w pielęgnacji. Ubocznym skutkiem tych kłopotów jest powstanie całej gałęzi wiedzy, zwanej „metodami programowania".

Alternatywą dla modelowania maszyny jest modelowanie samego problemu. Wczesne języki programowania, takie jak LISP czy APL, przyjmowały określony sposób widzenia świata („Wszystkie problemy są w ostateczności listami" albo „Wszystkie problemy są algorytmiczne"). PROLOG sprowadza wszystkie problemy do ciągów decyzji. Powstały języki przeznaczone do programowania opartego na ograniczeniach (ang. *constraint-based programming*) oraz do programowania wyłącznie poprzez manipulację symbolami graficznymi (te ostatnie okazały się zbyt restrykcyjne). Każde z tych podejść jest odpowiednie dla określonej klasy problemów, dla której zostały opracowane, jednak w innych przypadkach staje się nieefektywne.

Programowanie obiektowe idzie krok dalej, dostarczając programiście narzędzie do reprezentowania elementów w przestrzeni problemu. Reprezentacja ta jest dostatecznie ogólna, by nie ograniczała się do jakiegoś konkretnego typu problemów. Odwołujemy się do elementów w przestrzeni problemu, jak i do ich odpowiedników w przestrzeni rozwiązania, używając tego samego słowa — „obiekt" (oczywiście potrzebne są również obiekty, które nie mają odpowiedników w przestrzeni problemu). Pomysł polega na umożliwieniu programowi dostosowania się do specyficznego języka danego problemu poprzez dodanie nowych typów obiektów, dzięki czemu przy czytaniu kodu opisującego rozwiązanie natrafiamy na słowa wyrażające sam problem. Abstrakcja taka jest potężniejsza i bardziej elastyczna od tych, którymi dysponowaliśmy wcześniej[1]. Programowanie obiektowe pozwala więc opisywać problem raczej w kategoriach mu właściwych, nie zaś

[1] Niektórzy projektanci języków uznali, że programowanie obiektowe samo w sobie nie wystarcza, aby w prosty sposób rozwiązać wszystkie problemy programistyczne. Promują oni kombinację kilku różnych rozwiązań, zwaną *programowaniem z wieloma paradygmatami* (ang. *multiparadigm programming languages*). Patrz Timothy Budd, *Multiparadigm programming in Leda*, Addison-Wesley 1995.

w kategoriach komputera, na którym zostanie uruchomione rozwiązanie. Połączenie z komputerem jednak wciąż istnieje. Każdy obiekt przypomina mały komputer — ma swój wewnętrzny stan oraz zestaw operacji, które może wykonać, jeśli zostanie o to poproszony. Nie wydaje się to jednak złą analogią do obiektów świata rzeczywistego — one również mają pewną charakterystykę oraz zachowują się w określony sposób.

Alan Kay podsumował pięć podstawowych cech Smalltalka, pierwszego udanego języka obiektowego, a zarazem jednego z poprzedników Javy. Cechy te reprezentują czyste podejście obiektowe:

1. **Wszystko jest obiektem**. Można wyobrazić sobie obiekt jako wymyślną zmienną — taką, która nie tylko przechowuje dane, ale może także realizować żądania, czyli wykonywać na sobie pewne operacje. Teoretycznie, dowolne pojęcia ze świata rozwiązywanego problemu (psy, budynki, usługi itd.) można reprezentować w programie za pomocą obiektu.
2. **Program jest zbiorem obiektów, które poprzez wysyłanie komunikatów mówią sobie nawzajem, co robić**. „Wysłanie komunikatu" do obiektu to inaczej zażądanie wykonania przez niego pewnej operacji. Można też myśleć o komunikacie jako o wywołaniu funkcji przynależnej do konkretnego obiektu.
3. **Każdy obiekt posiada własną pamięć, na którą składają się inne obiekty**. Innymi słowy, nowy obiekt tworzymy, łącząc w jeden pakiet grupę już istniejących obiektów. Budujemy w ten sposób złożone programy, ukrywając jednocześnie ich złożoność za prostotą obiektów.
4. **Każdy obiekt posiada swój typ**. Mówiąc potocznie, każdy obiekt jest *instancją* pewnej *klasy*, gdzie „klasa" jest synonimem „typu". Najistotniejszą cechą wyróżniającą klasę jest odpowiedź na pytanie: „Jakie komunikaty można do niej wysyłać?".
5. **Wszystkie obiekty danego typu mogą otrzymywać te same komunikaty**. To stwierdzenie, jak się później przekonamy, może być nieco mylące. Ponieważ obiekt typu „okrąg" jest jednocześnie obiektem typu „figura", zatem „okrąg" może otrzymywać komunikaty odpowiednie dla „figury". Oznacza to, iż możliwe jest pisanie kodu odwołującego się do „figury", który będzie automatycznie obsługiwał wszystkie obiekty pasujące do opisu typu „figura". Ta *zastępowalność* jest jedną z najpotężniejszych idei programowania obiektowego.

Bardziej zwięzły i przejrzysty opis obiektu podaje Booch:

> *Obiekt ma stan, zachowanie i tożsamość.*

Oznacza to, że obiekt może posiadać dane wewnętrzne (określające jego stan), metody (stanowiące zachowanie obiektu) i każdy obiekt można w jednoznaczny sposób odróżnić od wszelkich innych obiektów — a konkretnie rzecz biorąc, z każdym obiektem jest skojarzony unikalny adres pamięci[2].

[2] Założenie to jest nieco ograniczające, gdyż można sobie wyobrazić obiekty istniejące na różnych komputerach oraz w różnych przestrzeniach adresowych, istnieje możliwość przechowywania obiektów na dysku. W takich przypadkach tożsamość obiektu należy określać nie na podstawie adresu pamięci, lecz w inny sposób.

Obiekt posiada interfejs

Pierwszym człowiekiem, który prowadził dokładne rozważania nad pojęciem *typu*, był prawdopodobnie Arystoteles — mówił on o „klasie ryb" oraz „klasie ptaków". Idea mówiąca, że wszystkie obiekty, będąc unikatowymi, należą jednocześnie do pewnych klas obiektów o wspólnych cechach i sposobach zachowania, została bezpośrednio wykorzystana w języku Simula 67. Podstawowym słowem kluczowym tego pierwszego obiektowego języka było słowo `class`, wprowadzające do programu nowy typ.

Jak sama nazwa wskazuje, Simula została zaprojektowana w celu tworzenia symulacji, takich jak klasyczny „problem kasjera bankowego". Mamy w nim do czynienia ze zbiorem kasjerów, klientów, kont, transakcji oraz sum pieniężnych — mnóstwem „obiektów". Obiekty, które są identyczne z wyjątkiem ich stanu podczas wykonywania programu, są pogrupowane w „klasy obiektów" — stąd właśnie wzięło się słowo kluczowe `class` (*klasa*). Tworzenie abstrakcyjnych typów danych (*klas*) jest jedną z fundamentalnych koncepcji programowania zorientowanego obiektowo. Abstrakcyjne typy danych zachowują się prawie tak samo, jak typy podstawowe: można tworzyć zmienne takiego typu (w języku programowania obiektowego nazywane obiektami lub egzemplarzami), a następnie manipulować nimi (proces ten nazywamy wysyłaniem komunikatów lub *żądań* — wysyłamy do obiektu komunikat, a on „domyśla się", co z nim zrobić). Składowe (elementy) danej klasy posiadają pewne cechy wspólne (np. każde konto ma saldo, każdy kasjer może przyjąć wpłatę), równocześnie jednak poszczególne składniki mają swój indywidualny stan — każde konto ma inne saldo, każdy kasjer inaczej się nazywa itd. A zatem każdy klient, konto, transakcja mogą być reprezentowane w programie komputerowym przez unikatowy byt. Byt ten jest obiektem, a każdy obiekt należy do określonej klasy, która definiuje jego cechy i zachowanie.

Tak więc, mimo iż w programowaniu obiektowym zajmujemy się tak naprawdę tworzeniem nowych typów, niemal wszystkie obiektowe języki programowania stosują słowo kluczowe *class*, nie *type*. Zatem kiedy widzimy słowo „typ", powinniśmy pomyśleć „klasa" — i na odwrót[3].

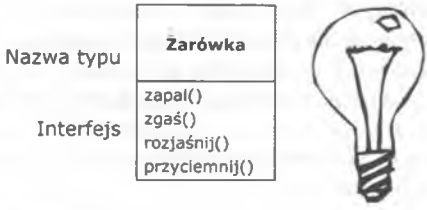
Ponieważ klasa opisuje zbiór obiektów o tej samej charakterystyce (danych składowych) oraz tych samych możliwych zachowaniach (możliwościach), jest typem danych tak samo prawdziwym, jak np. liczba zmiennoprzecinkowa, która także posiada pewne dane charakterystyczne (czyli swoją wartość) i zestaw dozwolonych na niej operacji (zachowań). Jedyna różnica polega na tym, że programista definiuje klasy odpowiadające raczej rozwiązywanemu problemowi — nie jest zmuszony do używania istniejących typów zaprojektowanych w celu reprezentowania właściwych maszynie jednostek przechowywania informacji. Dokonujemy więc rozszerzenia języka poprzez dodanie nowych typów danych, specyficznych dla naszych potrzeb. System programistyczny przyjmuje nasze klasy i zapewnia im taką samą „opiekę", jak typom wbudowanym, łącznie ze sprawdzaniem zgodności typów.

Programowanie obiektowe nie ogranicza się do budowania symulacji. Można nie zgadzać się z tezą, że każdy program jest w pewnym sensie symulacją rozwiązywanego przez siebie problemu, jednak mimo to prawdą pozostaje, że techniki obiektowe mogą z łatwością sprowadzić dużą klasę problemów do prostych rozwiązań.

[3] Niektórzy dokonują tu rozróżnienia, twierdząc, że typ określa tylko interfejs, natomiast klasa jest pewną implementacją tego interfejsu.

Gdy klasa jest już stworzona, możemy utworzyć dowolną liczbę jej obiektów, a następnie manipulować nimi tak, jak gdyby istniały w przestrzeni problemu. Jednym z podstawowych zadań w programowaniu obiektowym jest stworzenie jednoznacznego odwzorowania pomiędzy przestrzenią problemu a przestrzenią rozwiązania.

Jak jednak zmusić obiekt do zrobienia czegoś pożytecznego? Musi istnieć jakiś sposób zażądania od niego wykonania pewnej operacji, takiej jak zakończenie transakcji, narysowanie czegoś na ekranie czy też zmiana położenia przełącznika. Poza tym każdy obiekt może zrealizować tylko ściśle określone operacje. Żądania, jakie można skierować pod adresem danego obiektu, zdefiniowane są przez jego *interfejs*, a interfejs jest wyznaczony przez typ obiektu. Prostym przykładem może być reprezentacja żarówki:




```
Żarowka żr = new Żarowka();
żr.zapal();
```

Interfejs ustala, *jakie* żądania można kierować do danego obiektu, musi jednak istnieć kod realizujący te żądania. To on, wraz z ukrytymi danymi, składa się na *implementację*. Z punktu widzenia programowania proceduralnego nie jest to skomplikowane. Typ posiada powiązaną z każdym możliwym żądaniem funkcję, która jest wywoływana w chwili, gdy określone żądanie jest kierowane pod adresem jakiegoś obiektu. Proces ten jest określany jako „przesyłanie komunikatu" (żądanie) do obiektu i to obiekt wie, jak na dany komunikat zareagować (wykonuje kod).

W powyższym przykładzie typem (klasą) jest `Żarowka`, nazwą konkretnego obiektu typu `Żarowka` jest `żr`, a żądania, jakie `Żarowka` może obsłużyć, to: zapalenie, zgaszenie, rozjaśnienie i przyciemnienie. Stworzenie obiektu klasy `Żarowka` polega na zdefiniowaniu „referencji" (`żr`) dla tego obiektu, a następnie wywołaniu operatora `new` realizującego żądanie stworzenia nowego obiektu danego typu. W celu wysłania do obiektu komunikatu piszemy jego nazwę i łączymy ją za pomocą kropki z nazwą komunikatu, jaki chcemy wysłać. Z punktu widzenia użytkownika klas zdefiniowanych przez kogoś innego powiedzieliśmy już wszystko na temat programowania z wykorzystaniem obiektów.

Przedstawiony powyżej diagram jest zgodny z formatem zdefiniowanym przez UML (ang. *Unified Modeling Language* — ujednolicony język modelowania). Każda klasa jest reprezentowana przez prostokąt, którego górna część przeznaczona jest na jej nazwę, środkowa na pola danych (te, które chcemy opisywać), dolna zaś na *funkcje składowe* (funkcje należące do obiektu, to one otrzymują ostatecznie wszelkie komunikaty przesłane do obiektu). Bardzo często w diagramach projektowych języka UML przedstawiane są tylko nazwa i funkcje składowe klasy, dlatego środkowa część prostokąta nie jest rysowana. Jeśli interesuje nas tylko nazwa klasy — pomijamy także część dolną.

Obiekt dostarcza usługi

Pomimo prób tworzenia i zrozumienia projektu programu jednym z najlepszych sposobów pojmowania obiektów jest myślenie o nich jako o „dostawcach usług". Sam program świadczy pewne usługi dla użytkownika, realizując je przy użyciu usług oferowanych przez inne obiekty. Naszym celem jest stworzenie grupy obiektów (a jeszcze lepiej — odszukanie istniejącej biblioteki zawierającej takie obiekty) udostępniających usługi, które optymalnie nadają się do rozwiązania problemu.

Jednym ze sposobów, w jaki można rozpocząć realizację tego zadania, jest zadanie sobie pytania: „Gdybym, w jakiś magiczny sposób, mógł wyciągnąć je z kapelusza, jakie obiekty rozwiązałyby mój problem?". Na przykład, przypuśćmy, że piszemy program księgowy. Można by sobie wyobrazić obiekty zawierające stosowne, predefiniowane formularze, grupę obiektów realizujących odpowiednie obliczenia księgowe oraz obiekt obsługujący drukowanie czeków i faktur na wszelkiego typu drukarkach. Może niektóre z tych obiektów już istnieją, a jeśli nie, to jak powinny wyglądać? Jakie usługi powinny udostępniać *te* obiekty oraz jakich innych obiektów będą *one* potrzebować w celu realizacji swych zobowiązań? Postępując w ten sposób, można w końcu dojść do etapu, w którym będzie można stwierdzić, że „ten obiekt jest na tyle prosty, że można go szybko napisać" albo „jestem pewny, że taki obiekt już istnieje". To bardzo rozsądny sposób dekompozycji problemu na zbiór obiektów.

Wyobrażanie sobie obiektu jako dostawcy usług ma jeszcze jedną zaletę: pomaga w poprawieniu spójności obiektu. *Wysoka spójność* obiektu jest podstawowym wyznacznikiem jakości projektu oprogramowania. Oznacza ona, że różne aspekty fragmentów oprogramowania (takich jak obiekty, lecz może to także dotyczyć metod lub bibliotek obiektów) dobrze „pasują do siebie". Jednym z problemów, które się pojawiają podczas projektowania obiektów, jest wypełnienie nich zbyt wieloma możliwościami funkcjonalnymi. Na przykład, tworząc moduł drukujący, można dojść do wniosku, że konieczny będzie obiekt, który „wie wszystko" na temat formatowania i drukowania. Zapewne okaże się, że to zbyt wiele jak na jeden obiekt i że w rzeczywistości konieczne są trzy lub nawet więcej obiektów. Jeden z nich mógłby być katalogiem zawierającym wszystkie możliwe formaty czeków; z niego byłyby pobierane informacje dotyczące sposobu drukowania czeków. Kolejny obiekt, lub grupa obiektów, mógłby stanowić ogólny interfejs drukujący, który dysponowałby wiedzą na temat wszystkich rodzajów drukarek (lecz nie wiedziałby niczego na temat księgowości — taki obiekt bardziej nadaje się do kupienia niż do samodzielnego napisania). Trzeci obiekt mógłby korzystać z usług dwóch poprzednich, aby realizować zamierzone zadanie. A zatem każdy z obiektów dysponuje spójnym zbiorem udostępnianych usług. W poprawnym, obiektowo zorientowanym projekcie każdy obiekt realizuje dobrze jedno zadanie, lecz nie stara się robić zbyt wiele. Jak można się było przekonać, rozwiązanie takie nie tylko pomaga w odnajdywaniu obiektów, które można by wykorzystać (obiekt stanowiący interfejs drukowania), lecz także stwarza możliwość pisania obiektów nadających się do wielokrotnego wykorzystania w różnych programach (na przykład katalog czeków).

Traktowanie obiektu jako dostawcy usług jest podejściem, które wiele ułatwia i jest przydatne nie tylko podczas procesu projektowania, lecz także w sytuacjach, gdy inne osoby starają się przeanalizować kod lub powtórnie wykorzystać obiekty — jeśli można ocenić wartość obiektu na podstawie świadczonych przez niego usług, znacznie łatwiej można go dopasować do tworzonego projektu.

Ukrywanie implementacji

Warto dokonać rozróżnienia pomiędzy *twórcami klas* (ang. *class creators*, którzy definiują nowe typy danych) a *programistami-klientami* (ang. *client programmers*[4]; „konsumentami" klas, którzy wykorzystują te typy danych w swoich aplikacjach). Celem programisty-klienta jest zebranie zestawu klas-narzędzi gotowych do wykorzystania w szybkim tworzeniu aplikacji. Celem twórcy klas jest natomiast przygotowywanie takich klas, które udostępniają programiście-klientowi jedynie to, co dla niego niezbędne, a całą resztę trzymają w ukryciu. Dlaczego? Ponieważ to, co ukryte, nie może zostać wykorzystane przez programistę-klienta, a zatem twórca może zmienić niewidoczną część, nie przejmując się ewentualnym wpływem tych zmian na inne części. Ukryta część zwykle reprezentuje delikatne wnętrze obiektu, które mogłoby z łatwością zostać „popsute" przez nieuważnego lub niedoinformowanego programistę-klienta. Z tego powodu ukrywanie implementacji zmniejsza liczbę błędów w programach.

Nie sposób przecenić idei ukrywania implementacji. W każdej relacji istotne jest istnienie ograniczeń respektowanych przez wszystkie uczestniczące w niej strony. Gdy tworzymy bibliotekę, nawiązujemy równocześnie pewną relację z jej klientem, będącym również programistą, ale takim, który ma na celu „złożenie" aplikacji za pomocą naszej biblioteki, ewentualnie zbudowanie biblioteki większej. Gdy wszystkie składniki klasy są dostępne dla wszystkich, wtedy programista-klient może z klasą zrobić wszystko — nie istnieje żaden sposób na wymuszenie przestrzegania reguł. Nawet jeśli naprawdę nie chcemy, aby klient manipulował bezpośrednio niektórymi ze składników naszej klasy, bez mechanizmów kontroli dostępu nie istnieje sposób uniemożliwienia tego. Wszystko jest publiczne i widoczne dla całego świata.

Pierwszym uzasadnieniem kontroli dostępu jest chęć wymuszenia na kliencie trzymania rąk z daleka od rzeczy, których nie powinien dotykać — części niezbędnych do wykonywania wewnętrznych operacji naszego typu danych, nie będących jednak częścią interfejsu, którego potrzebują użytkownicy do rozwiązywania konkretnych problemów. Kontrola taka jest korzystna dla użytkowników, pozwala im bowiem łatwo odróżnić to, co dla nich istotne, od tego, co mogą zignorować.

Drugim powodem wprowadzenia kontroli dostępu jest umożliwienie projektantowi biblioteki wymiany wewnętrznych mechanizmów klasy bez zastanawiania się nad wpływem tej czynności na programistów-klientów. Można na przykład zaimplementować jakąś klasę w prymitywny sposób w celu uproszczenia pracy, a w późniejszym terminie (jeśli okaże się to konieczne) przepisać ją, aby działała szybciej. Jeżeli interfejs i implementacja są od siebie wyraźnie oddzielone i chronione, wtedy osiągnięcie tego nie jest trudne.

Java posiada trzy słowa kluczowe służące do ustanowienia rozgraniczeń w klasach. Są to: `public` (publiczny), `private` (prywatny) i `protected` (chroniony). Ich użycie i znaczenie jest dosyć oczywiste. Są to tzw. *specyfikatory dostępu* (ang. *access specifiers*) — określają, kto jest upoważniony do używania następujących po nich definicji. Specyfikator `public` oznacza, że definicje są dostępne dla każdego, natomiast `private`, że dostęp do nich posiada jedynie twórca danej klasy wewnątrz jej funkcji składowych. Jeżeli ktoś

[4] Termin ten autor zawdzięcza przyjacielowi, Scottowi Meyersowi.

próbuje używać prywatnych (private) elementów naszej klasy, powoduje tym samym wystąpienie błędu już w czasie kompilacji programu. Słowo kluczowe protected działa prawie tak samo jak private — różnica polega na tym, że klasy dziedziczące z naszej mają dostęp do elementów typu protected, nie mają natomiast do tych określonych jako private. Pojęcie dziedziczenia wprowadzę już niedługo.

Java posiada także „domyślny" tryb dostępu, który jest wykorzystywany, jeśli nie podamy żadnego ze wspomnianych specyfikatorów. Określany jest on czasem jako „pakietowy" (ang. *package access*), ponieważ do składowych pakietowych mogą się odwoływać inne klasy z tego samego pakietu, natomiast poza pakietem widziane są one jako prywatne.

Wielokrotne wykorzystanie implementacji

Stworzona i przetestowana klasa powinna (w sytuacji idealnej) stanowić użyteczny fragment kodu. Okazuje się jednak, iż osiągnięcie takiego stanu rzeczy nie jest tak proste, jak mogłoby się niektórym wydawać — stworzenie dobrego rozwiązania wymaga sporego doświadczenia i intuicji. Gdy jednak rozwiązanie takie już powstanie, wtedy aż prosi się o wielokrotne wykorzystanie. Umożliwienie takiego wielokrotnego wykorzystania kodu jest jedną z głównych zalet obiektowo zorientowanych języków programowania.

Najprostszym sposobem wykorzystania kodu klasy jest bezpośrednie użycie obiektu tej klasy, można jednak również umieścić obiekt wewnątrz nowej klasy. Nazywamy to „tworzeniem obiektu składowego". Nowa klasa może być zbudowana z dowolnej liczby obiektów (mogą one także należeć do różnych typów) połączonych w dowolne kombinacje potrzebne do osiągnięcia pożądanych możliwości. Ponieważ tworzymy nowe klasy, używając klas już istniejących, dlatego koncepcja ta nazywa się *kompozycją* (ang. *composition*); jeśli do kompozycji dochodzi dynamicznie (w czasie wykonania programu), mówimy o niej jako o *agregacji* (ang. *aggregation*). Kompozycja jest często określana jako relacja typu „posiada", tak jak w zdaniu „samochód posiada silnik".

(Na powyższym diagramie języka UML kompozycja zaznaczona jest za pomocą wypełnionego rombu. Zwykle będziemy używać prostszej formy — kreski niezakończonej rombem — dla oznaczenia asocjacji[5]).

Kompozycja umożliwia wysoki stopień elastyczności. Obiekty składowe naszej nowej klasy są zwykle prywatne, a więc niedostępne dla używających jej programistów-klientów. Można zatem składowe te podmieniać, nie zakłócając istniejącego kodu klienta. Obiekty składowe można także wymieniać w czasie wykonania, co pozwala na dynamiczną zmianę zachowania programu. Opisane dalej dziedziczenie nie zapewnia takiej elastyczności, ponieważ ograniczenia na klasy utworzone za jego pomocą nakładane są już w czasie kompilacji.

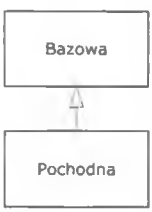
[5] Taki poziom szczegółowości jest wystarczający dla większości diagramów, zazwyczaj nie jest także konieczne rozróżnienie agregacji od kompozycji.

Ponieważ dziedziczenie jest tak istotne w programowaniu zorientowanym obiektowo, często jego znaczenie jest nadmiernie podkreślane, co może u początkującego programisty wytworzyć fałszywy pogląd, że należy je stosować wszędzie. W rezultacie tworzy on niceleganckie i zbytnio skomplikowane programy. Przy tworzeniu nowych klas powinniśmy raczej w pierwszej kolejności rozważać użycie kompozycji, która jest prostsza i bardziej elastyczna. Dzięki takiemu rozwiązaniu projekty programów będą znacznie czytelniejsze. Zidentyfikowanie sytuacji wymagających użycia dziedziczenia jest, gdy już zdobędzie się pewne doświadczenie, stosunkowo proste.

Dziedziczenie

Idea obiektu jest sama w sobie wygodnym narzędziem. Pozwala połączyć ze sobą dane i zestawy funkcji w celu reprezentowania określonego *pojęcia* z przestrzeni problemu bez używania specyficznego języka maszyny. Pojęcia takie są wyrażane jako podstawowe jednostki w języku programowania dzięki użyciu słowa kluczowego `class`.

Niedobrze by było, gdyby po rozwiązaniu wszystkich problemów związanych z tworzeniem klasy trzeba było tworzyć od podstaw nową klasę, posiadającą podobną funkcję. Łatwiejsze byłoby „sklonowanie" istniejącej klasy i wykonanie na powstałej kopii wymaganych przeróbek i rozszerzeń. To właśnie efektywnie osiągamy poprzez wykorzystanie *dziedziczenia* (ang. *inheritance*) — z tym, że gdy oryginalna klasa (nazywana klasą *bazową* lub *nadklasą*) zostanie zmodyfikowana, wtedy nasz „klon" (nazywany klasą *potomną*, klasą *pochodną* lub *podklasą*) odzwierciedli również te zmiany.



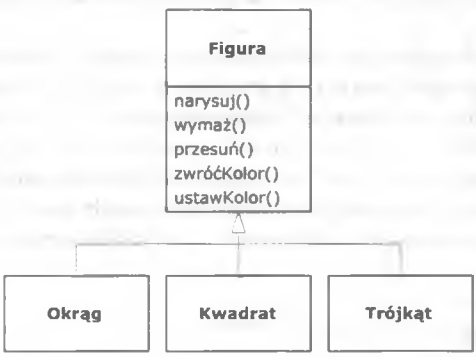
(Strzałka na powyższym diagramie UML jest skierowana od klasy pochodnej do bazowej. Jak później zobaczymy, klas pochodnych może być więcej).

Typ nie opisuje jedynie więzów nakładanych na pewien zbiór obiektów — wchodzi także w relacje z innymi typami. Dwa typy mogą posiadać wspólne cechy i zachowania, ale jeden może zawierać więcej danych niż drugi, lub obsługiwać więcej komunikatów (albo obsługiwać je inaczej). Dziedziczenie wyraża tego rodzaju podobieństwo między typami, używając pojęcia typów bazowych i typów pochodnych. Typ bazowy posiada te cechy i sposoby zachowania, które są wspólne dla wszystkich jego typów pochodnych. Tworzymy go do reprezentowania trzonu podstawowych idei, wspólnych dla pewnej grupy obiektów. Wywiedzione z niego typy pochodne reprezentują natomiast różne możliwości realizacji tych idei.

Rozważmy na przykład maszynę sortującą odpadki w celu odzysku surowców. Typem bazowym będzie „odpadek". Każdy odpadek posiada pewną wagę, wartość itd. Może być pocięty, przetopiony lub rozłożony na części. Dalej można wprowadzić bardziej

specyficzne typy śmieci — takie, które mają dodatkowe cechy (np. butelka posiada kolor) i sposoby zachowania (puszka aluminiowa może zostać zgnieciona, puszka stalowa reaguje na pole magnetyczne). Niektóre zachowania mogą dodatkowo ulec zmianie (np. wartość papieru zależy od jego rodzaju oraz stanu). Wykorzystanie dziedziczenia pozwala zbudować hierarchię klas wyrażającą rozwiązywany problem właśnie w kategoriach występujących w nim typów.

Jako drugi rozważymy klasyczny przykład z figurami geometrycznymi — może to być część systemu komputerowego wspomagania projektowania albo symulacji gry. Bazowym typem będzie tu „figura" posiadająca zawsze rozmiar, kolor, pozycję itp. Każda figura może zostać narysowana, wymazana, przesunięta, pokolorowana itd. Wywodzimy z niej (poprzez dziedziczenie) różne specyficzne typy figur: okręgi, kwadraty, trójkąty i inne, z których każdy może posiadać dodatkowe cechy i sposoby zachowania. Niektóre figury mogą być np. obrócone „do góry nogami". Niektóre zachowania mogą być różne, np. pole figury liczymy w różny sposób. Hierarchia typów wyraża zarówno podobieństwa, jak i różnice pomiędzy poszczególnymi rodzajami figur.



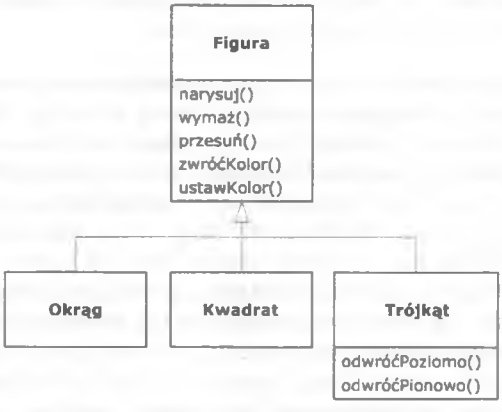
Opisanie rozwiązania w terminach problemu daje olbrzymie korzyści, ponieważ przejście od opisu problemu do opisu rozwiązania nie wymaga już zastosowania licznych modeli pośrednich. Gdy używamy obiektów, hierarchia klas stanowi model pierwotny, zatem przechodzimy wprost od opisu systemu w świecie rzeczywistym do opisu systemu w kodzie. Jednym z głównych problemów projektowania obiektowego jest zbyt prosta droga od początku do końca. Prostota często zbija początkowo z tropu umysły wytrenowane w poszukiwaniu skomplikowanych rozwiązań.

Poprzez dziedziczenie tworzymy nowy typ, który nie tylko zawiera wszystkie składowe (choć te zadeklarowane jako prywatne są ukryte i niedostępne), ale także, co ważniejsze, kopiuje interfejs klasy bazowej. A zatem wszystkie komunikaty, jakie można wysyłać do obiektów klasy bazowej, mogą być wysyłane również do obiektów klasy pochodnej. Ponieważ to zespół odbieranych komunikatów wyznacza typ, można powiedzieć, że *klasa pochodna jest tego samego typu co bazowa*. Odwołajmy się do naszego ostatniego przykładu: „Okrąg jest figurą". Ta osiągana przez dziedziczenie równoważność jest jednym z podstawowych etapów zrozumienia sensu programowania zorientowanego obiektowo.

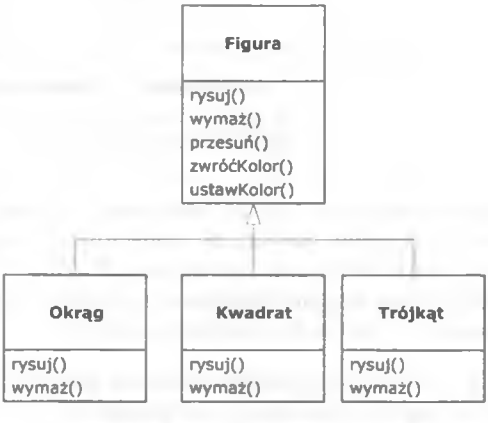
Ponieważ zarówno klasa bazowa, jak i pochodna mają ten sam interfejs, musi zatem istnieć jakaś związana z nim implementacja, tzn. musi istnieć kod wykonywany, gdy obiekt otrzyma określony komunikat. Jeżeli wszystko, co zrobimy, to stworzenie nowej klasy za

pomocą dziedziczenia, metody klasy bazowej stanowiące jej interfejs również przechodzą do klasy pochodnej — a zatem obiekty mają nie tylko ten sam typ co klasa bazowa, ale także zachowują się dokładnie tak samo, co nie jest szczególnie interesujące.

Istnieją dwa sposoby odróżnienia nowej klasy pochodnej od oryginalnej klasy bazowej. Pierwszy jest dosyć prosty: dodajemy do niej po prostu nowe metody. Nie są one częścią interfejsu klasy bazowej. Oznacza to, że klasa bazowa nie robiła wszystkiego, czego żądaliśmy, a zatem uzupełniliśmy jej zestaw metod. To proste i prymitywne użycie dziedziczenia jest, jak na razie, idealnym rozwiązaniem problemu. Warto jednakże zastanowić się, czy klasa bazowa nie potrzebuje tych dodatkowych metod. Taki proces iteracyjnego odkrywania właściwego projektu występuje często w programowaniu obiektowym.



Choć dziedziczenie wydaje się czasami (szczególnie w Javie, gdzie wprowadza się je za pomocą słowa kluczowego extends, czyli rozszerzać) sugerować konieczność dodania nowych metod do interfejsu, nie jest to do końca prawdą. Drugim — i do tego ważniejszym — sposobem, dzięki któremu możemy uczynić klasę pochodną różną od bazowej, jest *zmiana zachowania* istniejącej już metody bazowej, określana jako jej *przesłonięcie* (ang. *overriding*).

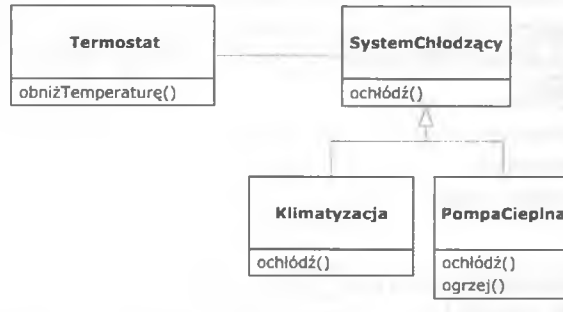


W celu przedefiniowania metody po prostu tworzymy w klasie pochodnej jej nową definicję. Mówimy: „Chcę tu wykorzystać tę samą metodę z interfejsu, ale dla nowego typu ma ona robić coś innego".

„Bycie czymś" a „bycie podobnym do czegoś"

W kwestii dziedziczenia może powstać pewna wątpliwość: czy nie należałoby ograniczyć się *jedynie* do przedefiniowywania metody klasy bazowej (powstrzymując się od dodawania nowych)? Oznaczałoby to, że klasa pochodna jest *dokładnie* tego samego typu co bazowa, ponieważ ma taki sam interfejs. W rezultacie obiekt klasy potomnej jest dokładnym substytutem obiektu klasy bazowej — możemy mówić o *czystej zastępowalności*, a całe to stanowisko jest określane mianem *zasady zastępowalności* (ang. *substitution principle*). Jest to w pewnym sensie idealny sposób traktowania dziedziczenia. Związek pomiędzy klasą bazową a pochodną określamy w tym przypadku często jako relację *bycia czymś* (ang. *is-a relationship*), ponieważ możemy powiedzieć „okrąg *jest* figurą". Sprawdzian poprawności użycia dziedziczenia polega na zbudowaniu podobnego zdania dla naszych klas i stwierdzeniu, czy ma ono sens.

W niektórych sytuacjach dodanie w typie pochodnym nowych elementów interfejsu jest jednak konieczne — takie jego rozszerzenie tworzy nowy typ. Może on być nadal używany zamiast typu bazowego, jednakże zastępowalność nie jest już idealna, ponieważ nowe funkcje nie są dostępne z poziomu bazowego. Ten rodzaj relacji możemy określić jako *bycie podobnym do czegoś*; nowy typ posiada cały interfejs starego typu, jednakże zawiera także inne metody, przez co nie można powiedzieć, że jest taki sam. Rozważmy na przykład klimatyzację. Przypuśćmy, że mamy zainstalowany w domu system chłodzący, a zatem interfejs pozwalający na kontrolę chłodzenia. Wyobraźmy sobie, że klimatyzacja psuje się i zastępujemy stary system nową pompą cieplną, która pozwala zarówno chłodzić, jak i ogrzewać. Zasada działania pompy cieplnej *jest podobna* do działania klimatyzacji, ale pompa może zrobić więcej od niej. Ponieważ system chłodzący jest zaprojektowany do kontroli chłodzenia, jest ograniczony do komunikowania się z chłodzącą częścią nowego obiektu. Interfejs tego obiektu został poszerzony, ale istniejący system nie zna niczego poza interfejsem oryginalnym.



Oczywiście po przestudiowaniu tego projektu staje się jasne, że klasa bazowa `SystemChłodzący` nie jest dostatecznie ogólna i powinna zostać zmieniona na „system kontroli temperatury", aby mogła obsługiwać również ogrzewanie — po czym zasada zastępowalności zaczęłaby znowu działać. Diagram ten przedstawia jednak sytuację, jaka może się zdarzyć w projektowaniu i w świecie rzeczywistym.

Po zapoznaniu się z zasadą zastępowalności można pomyśleć, że to rozwiązanie (czysta zastępowalność) jest jedynym słusznym, i faktycznie dobrze jest, gdy projekt działa w ten sposób. Z czasem jednak mogą wystąpić sytuacje, w których konieczność dodania nowych funkcji w interfejsie klasy pochodnej jest równie jasna. Po dokładnym zbadaniu rozróżnienie między tymi przypadkami powinno być oczywiste.

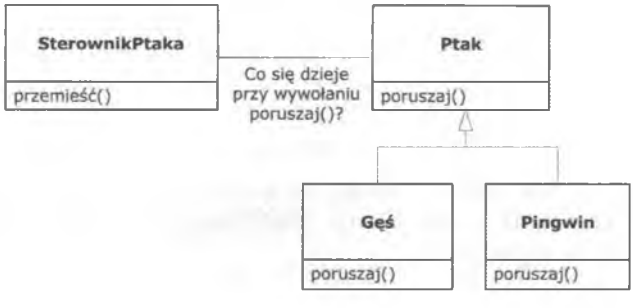
Wymienialność obiektów z użyciem polimorfizmu

Przy pracy z hierarchiami typów chcemy często traktować dany obiekt nie jako reprezentanta typu specjalizowanego, lecz raczej bazowego. Pozwala to na pisanie kodu niezależnego od konkretnego typu. W przykładzie z figurami funkcje manipulują nimi, nie zwracając uwagi na to, czy mają do czynienia z okręgami, kwadratami, trójkątami czy też takimi figurami, które nie zostały jeszcze nawet zdefiniowane. Wszystkie one mogą być narysowane, wymazane lub przesunięte, zatem funkcje te przesyłają po prostu komunikaty do obiektu typu figura, nie przejmują się sposobem, w jaki obiekt reaguje na te komunikaty.

Na kod taki nie ma wpływu dodawanie nowych typów, a dodawanie takie jest najpowszechniejszym sposobem rozszerzania programu obiektowego w celu obsłużenia nowych sytuacji. Możemy na przykład stworzyć nowy typ figury, zwany pięciokątem, nie modyfikując funkcji odnoszących się jedynie do ogólnych figur. Taka możliwość rozszerzania programu poprzez tworzenie nowych typów pochodnych jest ważnym sposobem hermetyzacji zmian, ponieważ znacząco ulepsza projekty, obniżając równocześnie koszty pielęgnacji oprogramowania.

Przy próbie traktowania typów pochodnych jako ich ogólnych typów podstawowych (np. okręgów jako figur, rowerów jako pojazdów, kormoranów jako ptaków itd.) powstaje jednak pewien problem. Jeżeli funkcja zamierza powiedzieć ogólnej figurze, aby się narysowała, ogólnemu pojazdowi, aby jechał lub ogólnemu ptakowi, aby się poruszył, kompilator nie może w czasie kompilacji określić, który kawałek kodu powinien zostać wykonany — funkcja rysowania może równie dobrze dotyczyć okręgu, kwadratu czy trójkąta, a obiekt wykona odpowiedni kod, wykorzystując swój właściwy, specyficzny typ.

Jeżeli nie musimy wiedzieć, który fragment kodu będzie wykonany, wtedy przy dodawaniu nowego podtypu kod przez niego wykonywany może być inny bez konieczności dokonywania zmian w wywołaniach metod. Kompilator nie może więc wiedzieć, który kawałek kodu zostanie wykonany. Co zatem robi? Na przykład na poniższym rysunku klasa `SterownikPtaka` pracuje wyłącznie z ogólnymi obiektami typu `Ptak`, nie znając ich konkretnych typów. Jest to wygodne z punktu widzenia klasy `SterownikPtaka`, ponieważ nie musi ona zawierać specjalnego kodu wyznaczającego dokładny typ lub zachowanie `Ptaka`, z jakim ma do czynienia. Jak więc się to dzieje, że choć `poruszaj()` wywoływane jest bez znajomości konkretnego typu `Ptaka`, to jednak ma miejsce właściwe zachowanie (Gęś biegnie, leci lub płynie, `Pingwin` biegnie lub płynie)?



Odpowiedzią jest podstawowa sztuczka programowania obiektowego: kompilator nie może wywołać funkcji w tradycyjny sposób. Wywołania funkcji generowane przez kompilatory języków niezorientowanych obiektowo używają tzw. *wczesnego wiązania* (ang. *early binding*) — terminu tego można nie znać, ponieważ wcześniej funkcji nie można było wywoływać inaczej. Znaczy to, że kompilator generuje wywołanie funkcji o określonej nazwie, program łączący (ang. *linker*) zamienia zaś to wywołanie na bezwzględny adres kodu, który ma zostać wykonany. W programowaniu obiektowym adres odpowiedniego fragmentu kodu nie może zostać wyznaczony aż do czasu wykonania, zatem przy wysyłaniu komunikatu do ogólnego obiektu konieczne jest inne rozwiązanie.

W celu rozwiązania tego problemu programowanie obiektowe wprowadza koncepcję *późnego wiązania* (ang. *late binding*). Przy wysyłaniu komunikatu do obiektu kod, który będzie wywołany, nie jest zdeterminowany aż do czasu wykonania. Kompilator upewnia się, że metoda istnieje, sprawdza typy argumentów i typ zwracanej wartości, nie wie jednak, jaki kod należy wykonać.

W celu przeprowadzenia późnego wiązania Java umieszcza zamiast bezwzględnego wywołania specjalny fragment kodu obliczający adres ciała metody na podstawie informacji przechowywanej w obiekcie (proces ten jest szczegółowo opisany w rozdziale 8., „Polimorfizm"). Każdy obiekt może zatem zachowywać się inaczej, w zależności od wyniku działania tego małego fragmentu kodu. Gdy wysyłamy komunikat do obiektu, obiekt decyduje, co z nim zrobić.

W niektórych językach musimy jawnie zadeklarować, że oczekujemy od funkcji elastyczności zapewnianej przez późne wiązanie (w języku C++ w tym celu wykorzystywane jest słowo kluczowe `virtual`). W językach tych wywołania funkcji składowych domyślnie *nie* są dynamicznie wiązane. Powodowało to problemy, dlatego w Javie dynamiczne wiązanie jest domyślne i nie musimy używać żadnych dodatkowych słów kluczowych, aby skorzystać z polimorfizmu.

Rozważmy przykład z figurami. Rodzina klas (bazujących na tym samym interfejsie) została przedstawiona nieco wcześniej na diagramie. Aby zademonstrować działanie polimorfizmu, chcielibyśmy napisać fragment programu ignorujący charakterystyczne dla typu szczegóły i wykorzystujący jedynie interfejs klasy bazowej. Kod ten będzie *niezależny* od informacji specyficznej dla typu, a przez to prostszy do napisania i łatwiejszy do zrozumienia. Poza tym, jeśli nowy typ, np. `Sześciokąt`, zostanie dodany poprzez dziedziczenie, kod, który napiszemy, będzie działał z tym nowym typem figury tak samo dobrze jak z istniejącymi wcześniej typami. Program jest zatem *rozszerzalny*.

Jeżeli napisze się w Javie (a niedługo nauczymy się to robić) następującą metodę:

```
void zróbCoś(Figura f) {
  f.wymaż();
  // ...
  f.narysuj();
}
```

komunikuje się ona z dowolną figurą, jest więc niezależna od konkretnego typu obiektu, który ma być rysowany i wymazywany. Jeżeli funkcję `zróbCoś()` wykorzystamy w jakimś innym fragmencie programu:

```
Okrąg o = new Okrąg();
Trójkąt t = new Trójkąt();
Linia l = new Linia();
zróbCoś(o);
zróbCoś(t);
zróbCoś(l);
```

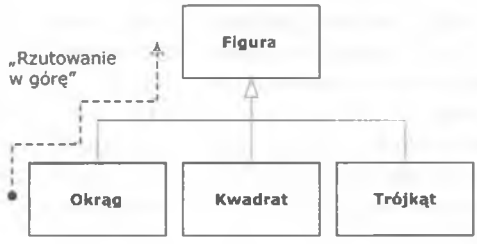
wywołania `zróbCoś()` będą automatycznie działać poprawnie, bez względu na dokładny typ obiektu.

W istocie jest to dosyć zaskakująca sztuczka. Rozważmy wiersz:

```
zróbCoś(o);
```

`Okrąg` jest przekazywany funkcji oczekującej argumentu typu `Figura`. Ponieważ `Okrąg` jest rodzajem figury, może więc być za taką uznawany przez metodę `zróbCoś()`. Znaczy to, że każdy komunikat, jaki `zróbCoś()` może wysłać do obiektu typu `Figura`, może być zaakceptowany przez `Okrąg`. To, co tutaj robimy, jest więc całkowicie logiczne i bezpieczne.

Proces taki, polegający na traktowaniu typu pochodnego tak, jakby był swoim typem bazowym, nazywamy *rzutowaniem w górę* (ang. *upcasting*). Rzutowanie używane jest tu w sensie dostosowywania do pewnej formy, *w górę* zaś odnosi się do sposobu, w jaki zwykle rysuje się diagramy dziedziczenia — z klasą bazową na szczycie i klasami pochodnymi rozwijającymi się w dół. Rzutowanie do typu bazowego oznacza zatem przesuwanie się w górę diagramu: jest więc rzeczywiście „rzutowaniem w górę".



Program zorientowany obiektowo zawiera w pewnych miejscach rzutowanie w górę, ponieważ w ten sposób uwalniamy się od konieczności znajomości dokładnego typu, z którym pracujemy. Spójrzmy na kod metody `zróbCoś()`:

```
f.wymaż();
  // ...
f.narysuj();
```

Jak widać, nie napisaliśmy tu: „Jeśli jesteś okręgiem, zrób to, jeśli jesteś kwadratem, zrób tamto, itd". Jeśli piszemy kod sprawdzający wszystkie typy, jakie może mieć `Figura`, staje się to nieeleganckie, a poza tym musi być zmieniane przy każdym dodaniu nowego rodzaju figury. W powyższym fragmencie mówimy po prostu: „Jeżeli jesteś figurą, to znaczy, że potrafisz wymazać się i narysować. Zrób więc to i zajmij się odpowiednio szczegółami".

W kodzie metody `zróbCoś()` imponuje fakt, że w jakiś sposób dzieje się to, co powinno. Wywołanie `narysuj()` dla obiektu `Okrąg` powoduje wykonanie innego kodu niż ten wykonywany przy wywołaniu tej samej metody dla `Kwadrat` albo `Linia`, jednak jeśli komunikat

narysuj() zostanie wysłany do anonimowej Figury, wtedy ma miejsce zachowanie poprawne dla rzeczywistego typu tej Figury. Jest to zadziwiające, ponieważ — jak już wspomniałem — kompilator Javy, kompilując zróbCoś(), nie wie, z jakimi dokładnie typami ma do czynienia. Normalnie oczekiwalibyśmy zatem, że użyje on wersji narysuj() i wymaż() zdefiniowanych dla bazowej klasy Figura, nie zaś specyficznych dla Okrąg, Kwadrat czy Linia. Mimo to dzięki polimorfizmowi dzieje się to, co powinno. Kompilator i system czasu wykonania zajmują się szczegółami — musimy jedynie wiedzieć, że to działa i, co istotniejsze, jak projektować z wykorzystaniem tego mechanizmu. Gdy wyślemy komunikat do obiektu, zrobi on to, co powinien, nawet jeśli użyte było rzutowanie w górę.

Hierarchia z pojedynczym korzeniem

Jedną z kwestii spornych, dotyczących programowania obiektowego, często poruszaną od czasu pojawienia się C++ jest to, czy wszystkie klasy powinny ostatecznie dziedziczyć po wspólnej klasie bazowej. W Javie (podobnie jak w niemal wszystkich językach obiektowych) odpowiedź brzmi „tak", nazwą tej wspólnej klasy bazowej jest zaś Object. Okazuje się, że hierarchia z pojedynczym korzeniem daje liczne korzyści.

W hierarchii z pojedynczym korzeniem wszystkie obiekty posiadają wspólny interfejs, więc w ostateczności wszystkie są tego samego typu. Alternatywa (dostarczana przez C++) polega na tym, że nie wiemy, czy wszystko ma ten sam fundamentalny typ. Z punktu widzenia wstecznej zgodności odpowiada to bardziej modelowi C, ale jest mniej ograniczające. Jednak jeśli chcemy programować w pełni obiektowo, musimy budować własną hierarchię, aby uzyskać taką samą wygodę, jaka jest wbudowana w inne języki obiektowe. Dodatkowo w każdej nowej bibliotece klas, jaką pozyskamy, użyty będzie jakiś inny, niezgodny interfejs. Dopasowanie tego nowego interfejsu do naszego projektu wymagać będzie wysiłku (a być może także wielokrotnego dziedziczenia). Czy dodatkowa „elastyczność" C++ jest tego warta? Jeżeli się jej potrzebuje — jeżeli zainwestowało się wiele w C — wtedy jest dosyć cenna. Jeżeli zaczynamy od zera, inne opcje, takie jak Java, mogą okazać się bardziej produktywne.

W hierarchii z pojedynczym korzeniem mamy gwarancję posiadania pewnych funkcji przez wszystkie obiekty. Wiemy, że pewne podstawowe operacje można wykonać na każdym obiekcie w systemie. Wszystkie obiekty można łatwo tworzyć w pamięci sterty, co znacząco uproszcza również przekazywanie argumentów.

Hierarchia o pojedynczym korzeniu w dużym stopniu ułatwia implementację *mechanizmu przywracania pamięci* (co jest jedną z ważniejszych zalet Javy wobec C++). Ponieważ informacja czasu wykonania o typie jest zagwarantowana w każdym obiekcie, nigdy nie pozostaniemy z obiektem, którego typu nie możemy określić. Jest to szczególnie istotne w przypadku operacji na poziomie systemu, takich jak obsługa wyjątków, oraz dla umożliwienia większej elastyczności w programowaniu.

Kontenery

Skoro zasadniczo nie wiemy, ile obiektów będzie nam potrzebnych do rozwiązania określonego problemu, ani jak długo obiekty te będą istnieć, nie wiemy również, w jaki sposób obiekty te przechowywać. Skąd mamy wiedzieć, ile miejsca dla nich przeznaczyć? Ta informacja jest niedostępna aż do czasu wykonania programu.

Rozwiązania większości problemów w projektowaniu obiektowym wydają się dziwnie proste: tworzymy nowy typ obiektów. W przypadku omawianego problemu tym nowym typem jest obiekt przechowujący referencje do innych obiektów. Moglibyśmy oczywiście uzyskać analogiczny efekt, używając *tablic* dostępnych w niemal każdym języku programowania. Dostajemy jednak więcej: nowy obiekt, zwany *kontenerem* (lub *kolekcją*, lecz słowo to jest używane w Javie w innym znaczeniu, w związku z tym w tej książce będzie używany termin „kontener"), będzie rozszerzał się, gdy będzie to konieczne, by przyjąć wszystko, co do niego wstawimy. Nie musimy więc wiedzieć, ile obiektów będzie przechowywanych w kontenerze. Po prostu tworzymy go i pozwalamy mu zająć się szczegółami.

Na szczęście każdy dobry język obiektowy dostarczany jest z zestawem kontenerów. W C++ zestaw taki jest częścią biblioteki standardowej i określany jest czasami akronimem STL (*Standard Template Library*). W języku Object Pascal kontenery stanowią część biblioteki VCL (*Visual Component Library*). Smalltalk posiada bardzo rozbudowaną bibliotekę kontenerów. Także Java posiada kontenery w swej bibliotece standardowej. W niektórych bibliotekach ogólny kontener uważany jest za wystarczająco dobry do wszystkich zastosowań, w innych zaś (np. w Javie) istnieją różne typy kontenerów dla różnych potrzeb: kilka różnych rodzajów klas `List` (służących do przechowywania sekwencji), klasy `Map` (zwane także *tablicami asocjacyjnymi*, które kojarzą jedne obiekty z innymi), klasy `Set` (do przechowywania niepowtarzających się elementów). Biblioteki kontenerów mogą również zawierać: kolejki, drzewa, stosy itd.

Z projektowego punktu widzenia wszystko, czego potrzebujemy, to kontener, którym można się posługiwać w celu rozwiązania problemu. Gdyby pojedynczy typ kontenera wystarczał, nie byłoby potrzeby posiadania różnych ich rodzajów. Istnieją dwa powody, dla których potrzebny jest wybór między kontenerami. Po pierwsze, kontenery posiadają różne interfejsy i zewnętrzne zachowania. Zachowanie się i interfejs stosu są różne od tych zapewnianych przez kolejkę, która z kolei różni się od zbioru czy listy. Określony interfejs może zapewniać bardziej elastyczne rozwiązanie naszego problemu niż inny. Po drugie, koszt wykonania tych samych operacji na różnych kontenerach jest inny. Najlepszy przykład stanowią `ArrayList` i `LinkedList`. Obie te klasy reprezentują proste listy elementów, mają identyczne interfejsy i cechują się podobnym zachowaniem zewnętrznym. Jednak określone operacje mogą mieć dla nich zasadniczo różny koszt. Dostęp swobodny do elementów w klasie `ArrayList` jest operacją wykonywaną w czasie stałym — trwa tak samo długo bez względu na to, do którego elementu się odwołujemy. Jednakże w przypadku `LinkedList` wybranie określonego elementu wymaga przejścia przez elementy poprzedzające go na liście, co jest operacją tym kosztowniejszą, im dalej na liście znajduje się element. Z drugiej strony, dodanie elementu w środku listy jest

znacznie tańsze w wypadku listy powiązanej (`LinkedList`). Te i inne operacje mają zróżnicowaną efektywność w zależności od wewnętrznej implementacji listy. W fazie projektowania moglibyśmy zacząć od wykorzystania listy powiązanej, po czym przy dostrajaniu efektywności przestawić się na wektor (`ArrayList`). Dzięki zapewnianej przez iteratory abstrakcji wpływ takiej zmiany na kod byłby minimalny.

Typy parametryzowane (typy ogólne)

W wersjach języka poprzedzających wydanie Java SE5 kontenery przechowywały elementy jednego, uniwersalnego typu: `Object`. Hierarchia z pojedynczym korzeniem implikuje, że wszystkie obiekty są reprezentantami klasy `Object`, a zatem kontenery mogące przechowywać obiekty tej klasy mogą przechowywać cokolwiek[6]. To właśnie dzięki temu kontenery nadają się do tak powszechnego stosowania.

Używając takiego kontenera, wstawiamy do niego po prostu referencje do obiektów, a później prosimy o ich zwrot. Jednak, skoro kontener przechowuje obiekty klasy `Object`, przy wstawianiu do kontenera dokonuje się rzutowania na tę klasę, przez co tracimy informację o konkretnym typie wstawianego obiektu. Wyjmując obiekt, otrzymujemy również referencję do typu `Object`, a nie do oryginalnego typu włożonego obiektu. Jak zatem zamienić ją na powrót na coś o interfejsie obiektu, który wstawiliśmy do kontenera?

I w tym przypadku potrzebujemy rzutowania, jednak nie będzie to już rzutowanie w górę hierarchii dziedziczenia dla uzyskania bardziej ogólnego typu — rzutujemy teraz w dół w celu uzyskania typu specjalizowanego. Ten rodzaj rzutowania nazywamy właśnie *rzutowaniem w dół* (ang. *downcasting*). Gdy rzutujemy w górę, wiemy na przykład, że „okrąg" jest rodzajem „figury", tak więc rzutowanie jest tu zawsze bezpieczne. Nie możemy jednak z góry przewidzieć, że dany obiekt typu `Object` jest z pewnością „okręgiem" czy „figurą" — trudno więc uznać ten rodzaj rzutowania za bezpieczny, chyba że w jakiś sposób poznamy właściwy typ obiektu, z którym mamy do czynienia.

Sytuacja nie jest jednak bardzo niebezpieczna — dokonując rzutowania na niewłaściwy typ, spowodujemy wystąpienie błędu czasu wykonania, zwanego *wyjątkiem* (ang. *exception*) — o wyjątkach dowiemy się więcej już niebawem. Mimo to przy wyjmowaniu z kontenera referencji do obiektów musimy, w celu dokonania prawidłowego rzutowania, pamiętać, jakiego właściwie te obiekty są typu.

Rzutowanie w dół, w połączeniu z koniecznością wykonywania testów czasu wykonania, powoduje pewien narzut czasowy w działającym programie oraz wymaga dodatkowego wysiłku od programisty. Można zapytać, czy nie dałoby się w jakiś sposób stworzyć kontenera „znającego" typ przechowywanych przez siebie obiektów i wyeliminować w ten sposób konieczność rzutowania oraz możliwość popełnienia błędów? Rozwiązaniem są *typy parametryzowane* (ang. *parametrized types*) — klasy, które kompilator automatycznie dostosowuje do pracy z podanymi typami-parametrami. Na przykład parametryzowany kontener może zostać przystosowany tak, aby akceptował tylko obiekty klasy `Figura` i zwracał także tylko `Figury`.

[6] Nie dotyczyło to typów elementarnych; mechanizm pakowania wartości elementarnych (skalarnych) w obiektach (ang. *autoboxing*) wprowadzony w Javie SE5 wyeliminował to ograniczenie niemal całkowicie. Wrócimy do tego w dalszej części książki.

Jedną z ważniejszych nowości w Javie SE5 jest właśnie obecność typów parametryzowanych, zwanych też *typami ogólnymi* (ang. *generics*). Parametryzację rozpoznajemy po obecności nawiasów kątowych przy nazwie klasy, z listą typów parametryzujących daną klasę. Na przykład obiekt klasa `ArrayList` (klasy kontenera) w wersji przystosowanej do przechowywania obiektów klasy `Figura` tworzy się tak:

```
ArrayList<Figura> figury = new ArrayList<Figura>();
```

Parametryzacja typów spowodowała liczne zmiany wielu komponentów biblioteki standardowej języka Java. Jak się niebawem okaże, dostępność typów ogólnych wpłynie na większość kodów przykładowych prezentowanych w tej książce.

Tworzenie obiektów i czas ich życia

Każdy obiekt, aby mógł istnieć, wymaga zasobów — przede wszystkim pamięci. Kiedy obiekt przestaje być potrzebny, należy po nim „posprzątać", czyli zwolnić przydzielone mu zasoby, aby można je było powtórnie wykorzystać. Zazwyczaj problem „posprzątania" po obiekcie nie wydaje się być szczególnym wyzwaniem — obiekt jest tworzony, stosowany tak długo, jak jest potrzebny, a następnie należy go usunąć. Jednak nietrudno spotkać sytuacje znacznie bardziej skomplikowane.

Załóżmy, że tworzymy system obsługi ruchu powietrznego dla lotniska. (Tego samego modelu można by także używać do zarządzania towarami w magazynie, w systemie wypożyczania kaset wideo lub firmie zajmującej się tresurą zwierząt). Na pierwszy rzut oka zadanie wydaje się proste: trzeba stworzyć kontener służący do przechowywania samolotów, a następnie umieścić w nim każdy samolot znajdujący się w kontrolowanym obszarze przestrzeni powietrznej. W ramach sprzątnięcia należy usunąć obiekt samolotu, który opuścił kontrolowany obszar.

Być może jednak dysponujemy innym systemem do rejestracji danych o samolotach; być może nie są to dane wymagające tak natychmiastowej uwagi, jak główne funkcje sterowania lotami. Być może jest to zapis planu lotów wszystkich małych samolotów startujących z lotniska. A zatem mamy drugi kontener dla małych samolotów i za każdym razem, gdy tworzony jest obiekt samolotu, jeśli jest to samolot mały, zapisujemy go także w tym drugim kontenerze. Następnie, podczas okresów bezczynności systemu, jakiś proces działający w tle wykonuje na tym kontenerze pewne operacje.

Teraz problem się skomplikował: w jaki sposób określić, kiedy można usunąć obiekt? Gdy obiekt nie jest już potrzebny w jednej części systemu, inne wciąż mogą z niego korzystać. Ten sam problem może się pojawiać w wielu różnych sytuacjach, a w systemach programistycznych wymagających jawnego usuwania obiektów (takich jak C++) może on stać się bardzo złożony.

Gdzie znajdują się dane obiektu i jak jest kontrolowany czas jego życia? Język C++ przyjmuje założenie, że najważniejszą sprawą jest kontrola efektywności, dlatego daje programiście wybór. W celu osiągnięcia maksymalnej szybkości czasu wykonania może on określić sposów przechowywania i czas życia obiektu na etapie pisania programu poprzez umieszczenie go na stosie (mówi się wtedy czasami o zmiennych *automatycznych*

albo *lokalnych*) albo w obszarze pamięci statycznej. Kładzie się w ten sposób nacisk na szybkie rezerwowanie i zwalnianie miejsca, poświęcając jednakże elastyczność, ponieważ musimy znać dokładną liczbę, czas życia oraz typ obiektów na etapie pisania programu. Jeżeli próbujemy rozwiązać bardziej ogólny problem, taki jak komputerowe wspomaganie projektowania, zarządzanie magazynem lub kontrola ruchu powietrznego, wtedy staje się to zbyt poważnym ograniczeniem.

Drugie rozwiązanie polega na dynamicznym tworzeniu obiektów w obszarze pamięci zwanym *stertą*. Stosując to rozwiązanie, nie wiemy aż do czasu wykonania, ilu obiektów potrzebujemy, jaki ma być czas ich życia oraz dokładny typ. Kwestie te są rozstrzygane w odpowiednim momencie podczas działania programu. Jeżeli potrzebujemy nowego obiektu, tworzymy go po prostu na stercie. Ponieważ pamięć jest zarządzana dynamicznie w czasie wykonania, zatem okres niezbędny do jej zarezerwowania na stercie jest o wiele dłuższy niż ten potrzebny do zrobienia tego samego na stosie (zarezerwowanie miejsca na stosie wiąże się często jedynie z pojedynczą instrukcją asemblerową do przesunięcia wskaźnika stosu w dół oraz drugą do przesunięcia go z powrotem w górę; czas konieczny do utworzenia miejsca na stercie zależy natomiast od konstrukcji mechanizmu obsługi pamięci). Czas dynamicznego przydziału w pamięci sterty jest zaś zależny od szczegółów mechanizmu obsługi pamięci w danym systemie.

Rozwiązanie dynamiczne bazuje na słusznym założeniu, że obiekty mogą być skomplikowane, a zatem narzut związany z samym znajdowaniem i zwalnianiem miejsca na te obiekty nie będzie miał istotnego wpływu na tworzenie obiektu. Ponadto zwiększona elastyczność tego rozwiązania ma podstawowe znaczenie dla rozwiązywania ogólnych problemów programistycznych.

Java stosuje wyłącznie drugie rozwiązanie[7]. Za każdym razem, gdy tworzymy obiekt, używamy słowa kluczowego `new`, aby utworzyć jego dynamiczny egzemplarz.

Inną sprawę stanowi czas życia obiektu. W językach pozwalających na tworzenie obiektów na stosie kompilator wyznacza czas trwania obiektu i może go automatycznie zniszczyć. Jeżeli jednak obiekt zostanie stworzony na stercie, wtedy kompilator nie zna czasu jego życia. W języku takim jak C++ musimy programowo określić, kiedy należy zniszczyć obiekt, co prowadzi do *wycieków pamięci*, jeśli nic robi się tego poprawnie (a jest to częsty przypadek w programach C++). Java posiada udogodnienie zwane *odśmiecaczem pamięci*, automatycznie wykrywającym, który obiekt nie jest już używany, a następnie niszczącym go. Odśmiecacz (ang. *garbage collector*) jest rozwiązaniem znacznie wygodniejszym, ponieważ redukuje liczbę zdarzeń, które musimy śledzić, oraz ilość kodu, jaki musimy napisać. Co ważniejsze, stanowi on znacznie wyższy stopień zabezpieczenia przed problemem wycieków pamięci (który zniweczył wiele projektów w C++).

W Javie problem zwalniania pamięci został oddelegowany do specjalnie zaprojektowanego odśmiecacza pamięci (choć nie obejmuje on pozostałych aspektów „sprzątania" po obiektach). Odśmiecacz „wie", kiedy obiekt nie jest już potrzebny, i zwalnia przydzieloną mu pamięć. Dzięki temu (oraz ze względu na fakt, że wszystkie obiekty dziedziczą po jednej klasie bazowej — `Object`, a wszystkie obiekty można tworzyć tylko w jeden sposób — na stercie) proces programowania w Javie jest znacznie prostszy niż w C++. Mniej tu decyzji do podjęcia i problemów do rozwiązania.

[7] Specjalny przypadek stanowią typy podstawowe, które poznamy później.

Obsługa wyjątków — eliminowanie błędów

Od początków istnienia języków programowania obsługa błędów była jednym z najtrudniejszych zadań. Z powodu trudności, jakich nastręcza zaprojektowanie dobrego schematu takiej obsługi, wiele języków po prostu ignoruje to zagadnienie, zrzucając odpowiedzialność na projektantów bibliotek. Proponują oni zwykle półśrodki działające poprawnie w wielu sytuacjach, będące jednak łatwe do obejścia (najczęściej po prostu przez ich zignorowanie). Podstawowym problemem większości schematów obsługi błędów jest to, że ich podstawą jest gotowość programisty do przestrzegania pewnej ustalonej konwencji, nieobsługiwanej w żaden sposób przez język. Gdy programista nie ma na to ochoty, a zdarza się to często, np. gdy się spieszy, może z łatwością o konwencji zapomnieć.

Mechanizm *obsługi wyjątków* (ang. *exception handling*) wiąże bezpośrednio obsługę błędów z językiem programowania, a czasem wręcz z systemem operacyjnym. Wyjątek jest obiektem, który jest „wyrzucany" z miejsca wystąpienia błędu, a następnie może zostać „przechwycony" przez odpowiednią *procedurę obsługi wyjątku* (ang. *exception handler*) — zaprojektowaną specjalnie do radzenia sobie z danym typem błędów. Obsługa wyjątków jest alternatywną ścieżką przepływu sterowania, wybieraną w przypadku, gdy coś pójdzie nie tak — dzięki temu nie musi się mieszać z kodem wykonywanym w normalnej sytuacji. Czyni to ten ostatni znacznie prostszym do napisania, ponieważ nie trzeba bez przerwy sprawdzać, czy nie wystąpiły jakieś błędy. Zgłaszanie („wyrzucanie") wyjątku różni się od ustawiania znacznika błędu czy zwracania ustalonej (oznaczającej błąd) wartości także tym, że w przeciwieństwie do nich nie może zostać zignorowane — mamy zatem gwarancję, że wyjątek zostanie w pewnym momencie obsłużony. Wyjątki dostarczają także godny zaufania sposób wyjścia ze złej sytuacji. Zamiast po prostu przerwać działanie programu, można często przywrócić warunki do jego dalszego wykonywania — napisane w ten sposób programy są znacznie solidniejsze.

Pod względem obsługi wyjątków Java znacząco różni się od innych języków programowania, gdyż mechanizmy te zostały wbudowane w język a programiści są zmuszeni do korzystania z nich. Jeśli tworzony kod nie będzie w poprawny sposób obsługiwać wyjątków, podczas jego kompilacji pojawią się błędy. Ta gwarantowana konsekwencja czasami może ułatwić obsługę błędów.

Warto zaznaczyć, że choć w językach zorientowanych obiektowo wyjątek jest zwykle reprezentowany przez obiekt, to jednak sam mechanizm obsługi wyjątków nie jest cechą przynależną jedynie tym językom — powstał od nich wcześniej.

Współbieżność

Jedną z podstawowych koncepcji programowania jest pomysł wykonywania kilku zadań w tym samym czasie. Liczne problemy programistyczne wymagają, aby program był w stanie przerwać swą bieżącą aktywność, załatwić jakąś inną sprawę i powrócić do głównego zadania. Próbowano wielu rozwiązań. Początkowo programiści posiadający

niskopoziomową wiedzę na temat maszyny pisali procedury obsługi przerwań, a zawieszanie głównego programu było inicjowane przez przerwania sprzętowe. Choć działało to zadowalająco, było jednak skomplikowane i nieprzenośne, czyniąc proces przenoszenia programu na nowy typ maszyny powolnym i kosztownym.

Przerwania są czasami niezbędne dla obsługi zadań o krytycznych ograniczeniach czasowych, jednakże w przypadku dużej klasy problemów chodzi nam o podzielenie zadania na wykonujące się niezależnie fragmenty, dzięki czemu program jako całość staje się bardziej efektywny. Wewnątrz programu te niezależnie wykonujące się fragmenty nazywamy wątkami, a cała idea nosi nazwę *współbieżności* (ang. *concurrency*) lub *wielowątkowości* (ang. *multithreading*). Typowym przykładem wykorzystania wielowątkowości jest interfejs użytkownika. Dzięki wątkom użytkownik może otrzymać szybką reakcję na naciśnięcie przycisku, nie będąc zmuszonym do oczekiwania na zakończenie wykonywania przez program jego aktualnego zadania.

Wątki są zwykle jedynie sposobem podziału czasu pojedynczego procesora. Jeżeli jednak system operacyjny obsługuje wiele procesorów, wtedy każdy wątek może zostać przydzielony innemu z nich, dzięki czemu będą się one wykonywać rzeczywiście równolegle. Jedną z zalet wielowątkowości na poziomie języka jest to, że programista nie musi interesować się, czy ma do dyspozycji wiele procesorów czy też tylko jeden. Program jest logicznie podzielony na wątki i jeżeli maszyna posiada wiele procesorów, wtedy działa szybciej bez żadnych poprawek.

Powyższe stwierdzenia mogą sugerować, że wielowątkowość jest prostym zagadnieniem. Jest jednak pewna pułapka — zasoby wspólne. Jeżeli więcej niż jeden wątek ma zamiar odwoływać się do tego samego zasobu, wtedy powstaje problem. Na przykład dwa procesy nie mogą równocześnie wysyłać informacji na drukarkę. W celu rozwiązania tego problemu zasoby, które mogą (tak jak drukarka) być dzielone, muszą być blokowane na czas użycia. A zatem wątek blokuje zasób, wykonuje swoje zadanie, po czym zwalnia blokadę, aby umożliwić innym wątkom skorzystanie z zasobu.

Wielowątkowość jest w Javie częścią języka; w Java SE5 doczekała się też istotnego wsparcia bibliotecznego.

Java i Internet

Jeśli tak naprawdę Java jest jeszcze jednym językiem programowania, można zapytać, czemu wobec tego jest tak ważna i dlaczego jest określana jako rewolucyjny krok w programowaniu? Z punktu widzenia tradycyjnego programowania odpowiedź nie jest od razu oczywista. Java jest bardzo użyteczna przy rozwiązywaniu tradycyjnych problemów programistycznych i rozwiązuje również problemy programowania dla sieci World Wide Web.

Czym jest sieć WWW?

Początkowo sieć WWW może się wydawać tajemnicza, razem z tymi wszystkimi surfowaniami, prezentacjami i stronami domowymi. Może warto przyjrzeć się z boku temu, czym tak naprawdę jest WWW? Jednak aby to zrobić, należy zrozumieć model systemów typu klient-serwer — kolejny pełen nieporozumień aspekt informatyki.

Przetwarzanie typu klient-serwer

Podstawą systemu klient-serwer jest założenie, że mamy centralne repozytorium informacji (jakiegoś rodzaju danych, często zapisanych w bazie danych), które chcemy rozsyłać na żądanie do pewnej grupy ludzi lub maszyn. Kluczem modelu klient-serwer jest centralne położenie składnicy informacji. Jeśli informacje zostaną zmienione, wszystkie zmiany zostaną przekazane ich odbiorcom. Całość: składnica informacji, oprogramowanie rozsyłające informacje oraz maszyna, na której znajduje się informacja, i jej oprogramowanie nazywane jest serwerem. *Klientem* nazywa się oprogramowanie znajdujące się na maszynie zdalnej, które komunikuje się z serwerem, pobiera informacje, przetwarza je i wyświetla na tej maszynie.

Podstawowa idea przetwarzania typu klient-serwer nie jest specjalnie skomplikowana. Problemy wynikają z tego, że mamy jeden serwer próbujący jednocześnie obsłużyć wiele klientów. Ogólnie, projektant wykorzystując systemy zarządzania bazami danych, stara się równomiernie rozmieścić dane w tabelach, aby osiągnąć jak największą użyteczność systemu. Dodatkowo systemy często pozwalają klientom na wprowadzanie nowych informacji do serwera. Oznacza to, że trzeba pilnować, aby nowe dane jednego klienta nie zamazały nowych danych innego lub żeby dane nie zaginęły w procesie dodawania ich do bazy (nazywa się to przetwarzaniem transakcyjnym). Kiedy oprogramowanie klienta się zmienia, musi być kompilowane, testowane i instalowane na maszynach klientów, co okazuje się bardziej skomplikowane i kosztowne, niż mogłoby się wydawać. Wielu problemów przysparza obsługa różnych typów komputerów i systemów operacyjnych. Na dodatek aktualny pozostaje problem wydajności: w każdej chwili mogą potencjalnie istnieć setki żądań skierowanych przez klientów do serwera, a zatem każde opóźnienie jest krytyczne. Aby ograniczyć te opóźnienia, programiści ciężko pracują nad tym, aby przenieść przetwarzane zadania na maszynę klienta, a czasem na inne maszyny po stronie serwera, składające się na tak zwaną *warstwę pośrednią* (ang. *middleware*). Warstwa pośrednia poprawia również możliwość konserwacji systemu.

Prosta idea rozsyłania ludziom informacji ma tak wiele poziomów złożoności, że cały problem może się wydawać bardzo zagmatwany. Jednocześnie jest bardzo ważny: przetwarzanie typu klient-serwer stanowi połowę wszystkich przedsięwzięć programistycznych. Odpowiada za wszystko, począwszy od przyjmowania zamówień i transakcji przy użyciu kart kredytowych, skończywszy na rozpowszechnianiu dowolnego typu danych — giełdowych, naukowych, rządowych. W przeszłości pojawiały się tylko indywidualne rozwiązania dla indywidualnych problemów — za każdym razem wynajdywano nowe. Były one trudne do tworzenia i używania, i za każdym razem użytkownik musiał się nauczyć obsługi nowego interfejsu. Problem klient-serwer wymaga rozwiązań na dużą skalę.

Sieć WWW jako gigantyczny serwer

Sieć WWW jest właściwie gigantycznym systemem typu klient-serwer. Nawet więcej — ponieważ wszystkie serwery i klienty współistnieją jednocześnie w tej samej sieci. Nie trzeba o tym wiedzieć, gdyż wszystko, o co trzeba się zatroszczyć w danym momencie, to połączenie i interakcja z jednym tylko serwerem (ale aby go znaleźć, można przeszukać pół świata).

Początkowo był to prosty proces jednokierunkowy. Na żądanie przesłane do serwera przekazywał on z powrotem plik, który był interpretowany i formatowany przez przeglądarkę na maszynie lokalnej klienta. Wkrótce ludzie zaczęli żądać czegoś więcej niż tylko dostarczania stron z serwera. Chcieli pełnych możliwości klient-serwer, tak aby klient mógł odsyłać informacje do serwera, na przykład zażądać konkretnych danych z serwera, dodać nowe informacje lub złożyć zamówienie (co wymagało większych zabezpieczeń niż te oferowane przez oryginalny system). Są to zmiany, jakie zaszły w rozwoju WWW.

Przeglądarka WWW — a właściwie sam pomysł, że ta sama informacja może być wyświetlona na komputerze dowolnego typu bez zmieniania jej — była wielkim krokiem naprzód. Jednakże przeglądarki były nadal prymitywne i nie spełniały stawianych im żądań. W szczególności nie były interaktywne i miały skłonność do blokowania zarówno serwerów, jak i Internetu, bowiem za każdym razem, kiedy konieczne było wykonanie czegoś, co wymagało przetwarzania, trzeba było wysłać informację z powrotem do serwera, aby dopiero tam została przetworzona. Mogło upłynąć wiele sekund lub minut, nim okazało się, że żądanie było błędnie sformułowane. Przeglądarka służyła jedynie do oglądania, więc nie mogła wykonywać nawet najprostszych zadań obliczeniowych (z drugiej strony była bezpieczna, ponieważ nie mogła na lokalnej maszynie wykonywać żadnych programów, które mogłyby zawierać błędy lub wirusy).

Aby rozwiązać ten problem, przyjęto kilka rozwiązań. Na początek standardy graficzne poszerzono tak, by umożliwić lepszą animację obrazu w przeglądarkach. Rozwiązaniem pozostałych problemów jest umożliwienie przeglądarce uruchamiania programów po stronie klienta. Nazywamy to *programowaniem po stronie klienta*.

Programowanie po stronie klienta

Pierwotny projekt architektury sieciowej typu serwer-przeglądarka zapewniał interaktywną zawartość, ale była ona w całości dostarczana przez serwer. Serwer produkował statyczne strony dla przeglądarki klienta, która je po prostu interpretowała i wyświetlała. Podstawowy HTML zawiera proste mechanizmy pobierania danych: pola tekstowe, pola wyboru, pola wielokrotnego wyboru, listy i listy rozwijane oraz przyciski, które można zaprogramować tylko do wyczyszczenia formularza lub wysłania (ang. *submit*) danych formularza z powrotem do serwera. To żądanie przechodzi przez *Common Gateway Interface* (uniwersalny interfejs komunikacyjny, w skrócie CGI) dostarczany przez wszystkie serwery WWW. Tekst wewnątrz żądania mówi CGI, co należy z nim zrobić. Najczęstszym działaniem jest wykonanie programu znajdującego się na serwerze w katalogu nazywanym zwykle *cgi-bin* (obserwując okienko adresu na górze przeglądarki, kiedy naciskasz przycisk na stronie sieciowej, możesz czasem zobaczyć *cgi-bin* pomiędzy innymi znajdującymi się tam śmieciami). Programy te mogą być pisane w większości języków. Najczęściej wybierany jest Perl, ponieważ jest przeznaczony do manipulowania tekstem, a programy w nim napisane są wykonywane przez interpreter, więc może być instalowany na dowolnych serwerach, niezależnie od procesora i systemu operacyjnego. Coraz większą popularność zdobywa jednak Python (mój ulubiony — patrz *www.Python.org*), głównie ze względu na swoje ogromne możliwości i prostotę.

Obecnie wiele dużych witryn internetowych zbudowano wyłącznie z wykorzystaniem CGI, za pomocą którego można zrobić praktycznie wszystko. Jednakże utrzymanie witryn zbudowanych na programach CGI może szybko stać się bardzo skomplikowane.

Problemem jest również długi czas reakcji. Odpowiedź programu CGI zależy od liczby danych, które muszą zostać przesłane, oraz od obciążenia serwera i Internetu (poza tym uruchamianie programu CGI jest na ogół wolne). Pierwsi projektanci Internetu nie przewidzieli, że jego przepustowość zostanie tak gwałtownie wyczerpana przez różnorodne aplikacje. Przykładowo, nie da się zrealizować procesu dynamicznego tworzenia grafiki, ponieważ plik GIF (*Graphics Interchange Format*) musi zostać stworzony i przesłany od serwera do klienta dla każdej wersji obrazu. Każdy bez wątpienia miał bezpośredni kontakt z czymś tak prostym, jak sprawdzenie danych w formularzu. Po naciśnięciu przycisku „Zatwierdź" dane są przesyłane z powrotem do serwera. Serwer uruchamia program CGI, który wykrywa błąd, formułuje stronę HTML informującą o błędzie, a następnie odsyła ją z powrotem. Trzeba wtedy cofnąć się do poprzedniej strony i spróbować ponownie. Jest to nie tylko wolne, ale i nieeleganckie.

Rozwiązaniem jest programowanie po stronie klienta. Większość komputerów, na których pracują przeglądarki, to potężne maszyny, będące w stanie wykonać ogromną ilość obliczeń, a przy pierwotnym sposobie pracy statycznego HTML-a czekające bezczynnie, aż serwer poda im następną stronę. Programowanie po stronie klienta sprawia, że przeglądarka WWW zostaje zaprzęgnięta do pracy, a dla użytkownika oznacza to o wiele bogatsze, interaktywne przeżycia podczas korzystania z takiej witryny.

Przy omawianiu programowania po stronie klienta problemem jest to, że nie różni się ono jakoś szczególnie od programowania w ogóle. Parametry są niemalże takie same, ale inna jest platforma: przeglądarka internetowa jest jak ograniczony system operacyjny. W końcu nadal trzeba programować, a to wiąże się z zawrotną liczbą problemów i rozwiązań stwarzanych przez programowanie po stronie klienta. Reszta tego podrozdziału przedstawia przegląd kwestii związanych z programowaniem po stronie klienta.

Moduły rozszerzające

Jednym z bardziej znaczących kroków w kierunku programowania po stronie klienta jest stworzenie modułów rozszerzających (ang. *plug-in*). Jest to sposób, w jaki program może dodać nowe funkcje do przeglądarki przez ściągnięcie kawałka kodu, który jest dołączany w odpowiednie miejsce przeglądarki. Mówi on przeglądarce: „od tej pory możesz wykonywać takie a takie nowe czynności" (moduł rozszerzający ładuje się jednorazowo). Dzięki rozszerzeniom przeglądarka zostaje wzbogacona o potężne możliwości, jednak pisanie ich nie jest zadaniem trywialnym i zapewne nikt nie chciałby tego robić w procesie budowania konkretnej strony. Moduły rozszerzające mają dużą wartość dla programowania klient-serwer, ponieważ pozwalają doświadczonemu programiście na stworzenie nowego języka programowania i dodanie go do przeglądarki bez konieczności uzyskania zgody jej twórcy. Zatem rozszerzenia dostarczają dodatkowe metody tworzenia nowych języków programowania po stronie klienta (jednak nie wszystkie języki są implementowane jako moduły rozszerzające).

Języki skryptowe

Wprowadzenie modułów rozszerzających zaowocowało eksplozją języków skryptowych. Kod źródłowy programu napisanego w języku skryptowym i działającego po stronie klienta jest osadzony wewnątrz strony HTML. Moduł interpretujący ten kod jest automatycznie aktywowany w momencie wyświetlania takiej strony. Języki skryptowe są na ogół łatwe

do zrozumienia i jako tekst są częścią kodu HTML. Program w takim języku jest tekstem stanowiącym fragment strony HTML, dlatego ładuje się z serwera jako część tego samego żądania, które wyświetla stronę. Wadą tego rozwiązania jest to, że kod źródłowy programu może być przeglądany (oraz skradziony) przez każdego. Jednak nie wydaje się to być zbyt wysoką ceną, ponieważ w językach skryptowych nie realizujemy wyszukanych zadań.

Jest taki język skryptowy, którego obsługi (i to bez dodatkowych modułów rozszerzających) można oczekiwać od większości nowoczesnych przeglądarek WWW: to JavaScript (który nie ma nic wspólnego z Javą — został tak nazwany, żeby się „załapać" na trochę marketingowego pędu związanego z Javą). Niestety, wersje języka zaimplementowane w różnych przeglądarkach mogą się znacznie od siebie różnić. Sytuację poprawiła nieco standaryzacja języka JavaScript w postaci *ECMAScript*, ale z kolei powszechne wdrożenie tego standardu wlecze się niemiłosiernie (ma w tym swój udział firma Microsoft, forsująca swój język skryptowy VBScript, wielce zresztą podobny do JavaScript). Zasadniczo należałoby przy programowaniu po stronie klienta ograniczyć się do najmniejszego wspólnego mianownika różnych implementacji JavaScript — tylko wtedy można mieć nadzieję bezproblemowego działania kodu w różnych przeglądarkach. Diagnostykę i wychwytywanie błędów w kodzie JavaScript można opisać jedynie jako udrękę. Dowodem stopnia złożoności problemu jest choćby to, że dopiero niedawno powstał pierwszy naprawdę rozbudowany projekt z użyciem JavaScript — mowa o Google GMail.

Widać z tego, że języki skryptowe, używane wewnątrz przeglądarek WWW, są przeznaczone do rozwiązywania specyficznego typu problemów, głównie do tworzenia bogatszych i bardziej atrakcyjnych graficznych interfejsów użytkownika (ang. *graphical user interface*, w skrócie GUI). Języki skryptowe mogą rozwiązać 80 procent problemów napotykanych przy programowaniu po stronie klienta. Również Twój problem może w całości mieścić się w tych 80 procentach, a ponieważ języki skryptowe pozwalają na łatwiejszą i szybszą produkcję stron WWW, powinieneś rozważyć ich użycie, zanim zwrócisz się ku bardziej złożonym rozwiązaniom, jak programowanie w języku Java.

Java

Jeśli języki skryptowe mogą rozwiązać 80 procent problemów programowania klient-serwer, co z pozostałymi 20 procentami — tymi naprawdę trudnymi? Obecnie najpopularniejszym rozwiązaniem jest Java. Jest to nie tylko potężny język programowania stworzony jako bezpieczny, przenośny i międzynarodowy. Java jest ciągle rozszerzana, by dostarczać nowe rozwiązania, które obsługują problemy uznawane za trudne w tradycyjnych językach, takie jak: wielowątkowość, dostęp do baz danych, programowanie sieciowe czy przetwarzanie rozproszone. Java umożliwia programowanie po stronie klienta poprzez *aplety* oraz technologię *Java Web Start*.

Aplet jest miniprogramem, który działa wyłącznie pod kontrolą przeglądarki. Jest on ładowany automatycznie jako element strony WWW (tak samo jak obrazek). Kiedy aplet zostanie aktywowany, wykona swój program. Jest to jego duży plus — zapewnia metodę automatycznego rozpowszechniania oprogramowania klienta z serwera w momencie, kiedy użytkownik go potrzebuje, a nie wcześniej. Użytkownik otrzymuje najnowszą wersję oprogramowania klienta bez borykania się z trudnościami ponownej instalacji. Java została zaprojektowana w taki sposób, że programista tworzy tylko jeden program, który automatycznie będzie działał na wszystkich komputerach wyposażonych w przeglądarki

z wbudowanym interpreterem Javy (można do nich zaliczyć większość maszyn). Ponieważ Java jest w pełni rozwiniętym językiem programowania, można więc obarczyć klienta dużą ilością pracy zarówno przed, jak i po wysłaniu żądania do serwera. Na przykład nie trzeba wysyłać żądania, aby odkryć, że źle podałeś datę lub inny parametr, a sam klient może szybko wykonać całą pracę związaną ze sporządzeniem wykresu, zamiast czekać, aż serwer przygotuje wykres i wyśle gotowy obrazek. Otrzymujesz natychmiastowy wzrost prędkości działania, a ogólny ruch sieci i obciążenie serwerów mogą zostać zredukowane, przyspieszając tym samym działanie całego Internetu.

Alternatywy

Gwoli szczerości, aplety Javy nie spełniły do końca pokładanych w nich pierwotnie oczekiwań. Kiedy Java ujrzała światło dzienne, wszyscy fascynowali się właśnie apletami jako wyczekiwanymi narzędziami programowania po stronie klienta, a więc zwiększaniem reaktywności i zmniejszeniem wymagań co do szybkości kanału transmisyjnego w aplikacjach internetowych. Apletom prorokowano wielką karierę.

W rzeczy samej w sieci WWW widuje się wielce zmyślne aplety, ale nigdy nie doszło do prawdziwej migracji do tej technologii. Największym problemem była najprawdopodobniej niechęć przeciętnych użytkowników do ściągania i instalowania środowiska wykonawczego Javy (JRE, od Java Runtime Environment) w postaci 10-megabajtowego pakietu. Los apletów mógł zostać przypieczętowany decyzją firmy Microsoft o niewłączaniu JRE do przeglądarki Internet Explorer. Tak czy inaczej aplety Javy nie zaistniały na większą skalę.

Mimo to aplety jako takie i technologia Java Web Start wciąż okazują się w pewnych sytuacjach przydatne. Znajdują zastosowanie wszędzie tam, gdzie zachodzi potrzeba kontrolowania maszyn użytkowników, na przykład w obrębie systemu informatycznego korporacji; służą tam do rozprowadzania i aktualizacji aplikacji klienckich przy znacznej oszczędności czasu oraz zasobów ludzkich i finansowych — widocznej zwłaszcza tam, gdzie aktualizacje są częste.

W rozdziale „Graficzne interfejsy użytkownika" przyjrzymy się nowej, obiecującej technologii *Flex* firmy Macromedia, która pozwala na tworzenie odpowiedników apletów, tyle że opartych na technologii Flash. Ponieważ jakieś 98 procent przeglądarek WWW dysponuje odtwarzaczami formatu Flash (dotyczy to przeglądarek dla wszystkich popularnych systemów operacyjnych), można go uznać za powszechnie przyjęty standard. Instalacja i aktualizacje odtwarzacza Flash odbywają się szybko i prosto. Stosowany tu język ActionScript bazuje na języku ECMAScript, przez co łatwo się do niego przyzwyczaić, ale Flex pozwala na programowanie bez martwienia się o charakterystykę przeglądarki, co czyni go znacznie atrakcyjniejszym od JavaScriptu. W dziedzinie programowania po stronie klienta jest to z pewnością alternatywa godna rozważenia.

.NET i C#

Od dłuższego czasu głównym konkurentem apletów Javy były komponenty ActiveX, choć wymagały one, aby klient działał w systemie operacyjnym Windows. Następnie Microsoft stworzył godnego rywala dla Javy — platformę *.NET* oraz język *C#*. Platforma .NET jest mniej więcej tym samym co *wirtualna maszyna Javy* oraz wszystkie biblioteki tego języka, a podobieństwa języków C# i Java są oczywiste. Bez wątpienia jest to najlepsza

robota, jaką Microsoft wykonał w dziedzinie języków programowania i środowisk programistycznych. Oczywiście Microsoft miał znaczącą przewagę, gdyż mógł przeanalizować, co w Javie było dobre a co złe, i bazować na tej wiedzy. Jednak została ona wykorzystana z doskonałym rezultatem. Po raz pierwszy od momentu pojawienia się Java zyskała godnego siebie konkurenta; dzięki temu z kolei projektanci języka Java przysiedli fałdów i, przyglądając się C# oraz przyczynom, dla których programiści mogą wybierać go, a nie Javę, odpowiedzieli rozszerzeniem Javy w postaci wydania Java SE5 z jego fundamentalnymi usprawnieniami.

Aktualnie głównym słabym punktem oraz pytaniem dotyczącym platformy .NET jest to, czy Microsoft zezwoli na przeniesienie go w *całości* na inne platformy systemowe. Microsoft twierdzi, że nie powinno to być żadnym problemem, a projekt Mono (*www.go-mono.com*) stanowi częściową implementację platformy .NET działającą w systemach Linux. Jednak do momentu zakończenia prac nad pełną implementacją i podjęcia przez Microsoft decyzji dotyczącej wszystkich elementów platformy .NET wykorzystanie jej jako rozwiązania międzysystemowego jest ryzykowne.

Internet kontra intranet

Sieć WWW jest najogólniejszym rozwiązaniem problemu klient-serwer, więc wydaje się sensowne użycie tej samej technologii do podzbioru tego problemu — w szczególności klasycznego problemu klient-serwer *wewnątrz* firmy. Przy tradycyjnym podejściu klient-serwer występują trudności z powodu wielu typów komputerów po stronie klientów oraz kłopoty z instalacją nowego oprogramowania. Obydwa problemy są dobrze rozwiązywane przez przeglądarkę WWW i programowanie po stronie klienta. Kiedy technologia WWW jest używana w sieci informacyjnej ograniczonej do konkretnej firmy, określa się ją mianem intranetu. Intranet dostarcza o wiele wyższy poziom bezpieczeństwa niż Internet, ponieważ można fizycznie kontrolować dostęp do firmowych serwerów. Jeśli wziąć zaś pod uwagę szkolenie pracowników, wydaje się, że kiedy już zrozumieją ogólne zasady pracy z przeglądarką, jest im dużo łatwiej radzić sobie z różnicami między działaniem stron i apletów, a czas potrzebny do nauczenia się nowych systemów jest krótszy.

Problem bezpieczeństwa doprowadził do jednego z podziałów powstałych automatycznie w świecie programowania typu klient-serwer. Jeśli program działa w Internecie, nie wiemy, na jakiej platformie będzie pracował, a jednocześnie zwracamy szczególną uwagę, aby nie rozpowszechniać kodu zawierającego błędy. Potrzebujemy czegoś tak przenośnego i bezpiecznego, jak język skryptowy lub Java.

Pracując w intranecie, napotykamy zestaw innych ograniczeń. Nie jest rzeczą niezwykłą, że wszystkie maszyny będą pracować na platformie Intel-Windows. W intranecie odpowiadamy za jakość własnego kodu i możemy naprawiać błędy zaraz po ich wykryciu. W dodatku często trzeba wykorzystać kod pozostały po wcześniejszych, tradycyjnych implementacjach systemu. Trzeba wtedy fizycznie instalować programy klientów przy każdorazowym uaktualnieniu. Czas tracony na instalowanie uaktualnień jest najczęstszym powodem przejścia na korzystanie z przeglądarki, ponieważ tutaj uaktualnienia są niewidoczne i automatyczne. Jeśli jesteś zaangażowany w tego typu projekt intranetowy, najrozsądniejszym rozwiązaniem jest obranie najprostszej drogi, umożliwiającej wykorzystanie istniejącej bazy kodu zamiast przepisywania programów ponownie w nowym języku.

W przypadku tak konsternującego bogactwa rozwiązań problemu klient-serwer najlepszym wyjściem jest analiza kosztów i korzyści. Rozważ ograniczenia postawionego problemu i odszukaj najkrótszą ścieżkę do rozwiązania. Ponieważ programowanie po stronie klienta nadal pozostaje programowaniem, zawsze dobrym pomysłem jest przyjęcie podejścia najszybciej prowadzącego do rozwiązania. Jest to agresywna postawa przygotowująca na spotkanie z problemami nieuniknionymi przy tworzeniu oprogramowania.

Programowanie po stronie serwera

W dotychczasowej dyskusji pomijany był temat programowania po stronie serwera. Co się dzieje w momencie wysłania żądania do serwera? Najczęstszym żądaniem jest proste: „Wyślij mi ten plik". Następnie przeglądarka interpretuje otrzymany plik w odpowiedni sposób: jako stronę HTML, obrazek, aplet Javy, skrypt itd.

Bardziej skomplikowane żądania kierowane do serwera wymagają na ogół komunikacji z bazą danych. W wielu przypadkach wymaga to wykonania złożonego zapytania na bazie danych, które serwer formatuje jako stronę HTML i odsyła z powrotem do klienta (oczywiście jeśli klient ma większe możliwości — dzięki Javie lub językom skryptowym — to surowe dane mogą być przesłane i sformatowane po stronie klienta, co jest szybsze i mniej obciąża serwer). Podobnie rejestracja użytkownika w chwili dołączenia do grupy dyskusyjnej lub złożenia zamówienia wymaga wprowadzenia zmian w bazie danych. Te wszystkie żądania muszą zostać przetworzone przez jakiś program działający po stronie serwera, co ogólnie określane jest jako programowanie po stronie serwera. Tradycyjnie programy CGI działające po stronie serwera były tworzone przy użyciu Perla, Pythona lub C++, ale pojawiały się też bardziej wyszukane systemy. Weźmy pod uwagę np. serwery sieciowe wykorzystujące Javę. Umożliwiają one wykonanie całości programowania po stronie serwera przez pisanie *serwletów*. Serwlety i ich pochodna JSP eliminują problemy związane z różnorodnym poziomem zaawansowania różnego typu przeglądarek. Z tego powodu wiele firm tworzących strony WWW przechodzi na Javę. (Te zagadnienia zostały opisane w książce *Thinking in Enterprise Java — zobacz www.MindView.net*).

Większość szumu wokół Javy związana była z apletami. W rzeczywistości Java jest językiem programowania ogólnego przeznaczenia, który może rozwiązywać dowolny rodzaj problemów. Siłą Javy jest nie tylko jej przenośność, ale także możliwości programistyczne, niezawodność, powszechność, dostępność bibliotek standardowych i licznych łatwo dostępnych i żywiołowo rozwijających się bibliotek dodatkowych.

Podsumowanie

Wszyscy wiemy, jak wygląda program proceduralny: definicje danych i wywołania funkcji. Aby odgadnąć znaczenie takiego programu, trzeba się troszkę napracować, prześledzić wywołania funkcji i zbadać niskopoziomowe pojęcia, aby stworzyć własny myślowy model. Z tego powodu potrzebujemy reprezentacji pośrednich przy projektowaniu programu proceduralnego — programy te same w sobie mogą być niezrozumiałe, ponieważ środki wyrazu są skierowane bardziej na komputery niż na rozwiązywany problem.

Ponieważ programowanie obiektowe dodaje wiele pojęć do tych, które można znaleźć w językach proceduralnych, może się wydawać naturalnym założeniem, że wynikowy program w języku Java może być o wiele bardziej skomplikowany, niż jego równoważnik w języku proceduralnym. Tutaj spotyka nas miła niespodzianka: dobrze napisany program jest na ogół o wiele łatwiejszy do zrozumienia niż odpowiadający mu kod proceduralny. To, co widzimy, to definicje obiektów reprezentujących pojęcia z przestrzeni problemu (a nie kwestie związane z reprezentacją komputerową) oraz komunikaty wysyłane do tych obiektów, reprezentujące działania w przestrzeni problemu. Jedną z przyjemnych stron programowania zorientowanego obiektowo jest to, że dobrze zaprojektowany program można zrozumieć, czytając jego kod. Najczęściej kod jest też o wiele krótszy, ponieważ wiele problemów zostało rozwiązanych przez zastosowanie kodu z istniejących bibliotek.

Programowanie obiektowe i Java nie muszą być dobre dla każdego. Ważne jest, by ocenić swoje potrzeby i zdecydować, czy Java w sposób optymalny je zaspokaja, czy też lepiej będzie użyć innego języka programowania (wliczając obecnie używany). Jeśli wiadomo, że w przewidywalnej przyszłości stawiane wymagania będą bardzo specjalne i jeśli dodatkowo pojawiają się specyficzne ograniczenia, których Java nie wyeliminuje, lepiej będzie zbadać rozwiązania alternatywne (w szczególności polecam zwrócenie uwagi na język Python; patrz *www.Python.org*). Jeśli ostatecznie wybór padnie na Javę jako preferowany język programowania, będzie to świadoma decyzja podjęta po rozważeniu innych opcji.

Rozdział 2.

Wszystko jest obiektem

Kiedy mówimy różnymi językami, postrzegamy nieco inne światy

— Ludwig Wittgenstein (1889 – 1951).

Mimo że Java bazuje na języku C++, jest językiem znacznie bardziej zorientowanym obiektowo.

C++ i Java są językami hybrydowymi. Projektanci Javy doszli jednak do wniosku, że element hybrydyzacji nie jest tak istotny, jak to było w C++. Język hybrydowy pozwala programiście na używanie wielu stylów programowania równocześnie — przyczyną tego, że C++ pozostał hybrydowy, była chęć zachowania zgodności z jego poprzednikiem — językiem C. Ponieważ C++ jest nadzbiorem C, ciągle zawiera wiele niepożądanych elementów językowych, które komplikują niektóre aspekty C++.

Java zakłada, że programiści chcą pisać programy jedynie w sposób zorientowany obiektowo. Znaczy to, że przed rozpoczęciem programowania należy przestawić swój sposób myślenia na obiektowy (jeżeli jeszcze tego nie uczyniliśmy). Umożliwi to programowanie w języku, który jest łatwiejszy zarówno do nauki, jak i w użyciu niż wiele innych języków obiektowych. W tym rozdziale poznamy podstawowe części składowe programu napisanego w Javie oraz przekonamy się, że wszystko w Javie jest obiektem.

Dostęp do obiektów poprzez referencje

Każdy język programowania posiada określony sposób dostępu do danych. Czasami programista jest zmuszony stale kontrolować, jakie dokładnie operacje są wykonywane: czy odwołujesz się do obiektów bezpośrednio czy może używasz jakiegoś rodzaju pośredniej reprezentacji (wskaźnik w C lub C++), który wymaga stosowania specjalnej składni?

W Javie wszystko to zostało uproszczone. Dosłownie wszystko jest tu bowiem traktowane jako obiekt, a więc mamy jeden zwarty zapis, którego używamy wszędzie. Pomimo traktowania wszystkiego jak obiektu, identyfikator, którym się posługujemy, jest tak naprawdę

„odwołaniem" (referencją) do obiektu[1]. Można to wytłumaczyć na przykładzie telewizora (obiekt) sterowanego pilotem (referencja). Tak długo, jak posiadamy referencję, mamy połączenie z telewizorem. Jeżeli ktoś poprosi: „Zmień program" lub „Przycisz", to posługujemy się właśnie referencją, która z kolei zmienia stan obiektu. Jeżeli chcemy chodzić po pokoju i nadal mieć kontrolę nad telewizorem, wystarczy wziąć ze sobą referencję, a nie cały telewizor.

Taki zdalny panel może też istnieć samodzielnie, bez telewizora. Oznacza to, że jeśli istnieje referencja, wcale nie musi być do niej przypisany jakiś obiekt. Zatem, jeżeli chcemy przechowywać słowo lub zdanie, to tworzymy referencję typu `String`:

```
String s;
```

Ale przecież stworzyliśmy tu *jedynie* referencję, a nie obiekt. Jeżeli chcielibyśmy przesłać wiadomość do zmiennej `s`, otrzymalibyśmy komunikat o błędzie, gdyż `s` w istocie nie jest z niczym powiązana (nie ma telewizora). Bezpieczniejszą praktyką jest zainicjowanie referencji przy każdym jej tworzeniu:

```
String s = "asdf";
```

Powyższy przykład wykorzystuje jednak specyficzną cechę Javy: łańcuch tekstowy może być zainicjowany poprzez podanie tekstu w cudzysłowie. Normalnie należy użyć ogólnego sposobu inicjalizacji obiektu.

Wszystkie obiekty trzeba stworzyć

Kiedy tworzymy referencję, chcemy powiązać ją z nowym obiektem. Zazwyczaj stosuje się w tym celu słowo kluczowe `new`. Słowo `new` mówi: „Utwórz mi jeden nowy obiekt danego typu". A więc, jeśli posłużymy się poprzednim przykładem, może to wyglądać tak:

```
String s = new String("asdf");
```

Nie tylko znaczy to: „Utwórz nowy `String`", ale zawiera także informację, *jak* stworzyć `String` na podstawie podanego łańcucha tekstowego.

[1] Jest to kwestia sporna, niektórzy bowiem twierdzą: „To zrozumiałe, że jest to wskaźnik", ale tutaj implementacja jest inna. Odwołania w Javie są nawet bardziej spokrewnione w składni z referencjami C++ niż ze wskaźnikami. W pierwszym wydaniu tej książki zdecydowałem się na użycie nowego terminu „uchwyt", ponieważ referencje w Javie i C++ różnią się zasadniczo pod kilkoma względami. Osobiście wyszedłem od C++ i nie chciałem plątać pojęć programistom C++, którzy — jak przypuszczam — stanowią największe audytorium Javy. W drugim wydaniu zdecydowałem się jednak na użycie określenia „referencja" ze względu na to, że jest bardziej rozpowszechnione i ktoś przechodzący od C++ mógłby mieć sporo do przyswojenia, a tak ma możliwość szybkiego „wskoczenia" do nowego języka. Są jednak ludzie, którzy nie zgadzają się z określeniem „referencja". Czytałem w pewnej książce, że jest to „zupełnie błędne stwierdzenie, że Java obsługuje przekazywanie poprzez referencje", ponieważ identyfikatory obiektów Javy (nawiązując do tamtego autora) są faktycznie „referencjami do obiektów". Dalej (jak twierdzi tamten autor) wszystko jest *faktycznie* przekazywane przez wartość. Tak więc nie przekazujemy przez referencję, ale „przekazujemy referencję do obiektu przez wartość". Ktoś mógłby dyskutować na temat szczegółowości tak zagmatwanych objaśnień, ale myślę, że moje podejście upraszcza rozumienie pojęcia, nie krzywdząc nikogo (no dobrze, puryści językowi mogą posądzić mnie o kłamstwa, lecz odpowiem, że jest to jedynie abstrakcyjne ujęcie).

Oczywiście typ `String` nie jest jedynym istniejącym. Java dostarcza bowiem bardzo wiele gotowych typów. Ważniejsza jest jednak możliwość tworzenia własnych typów — jest zasadniczą sprawą podczas programowania w Javie i tym, czego będziesz się uczył w dalszej części tej książki.

Gdzie przechowujemy dane

Przydatne jest zrozumienie kilku aspektów dotyczących tego, gdzie umieszczane są dane podczas działania programu, a szczególnie, jak przydzielana jest pamięć. Jest bowiem pięć różnych miejsc, w których przechowuje się dane:

1. **Rejestry**. Jest to najszybciej dostępna pamięć z uwagi na to, że istnieje w innym miejscu niż pozostałe jej rodzaje — wewnątrz procesora. Liczba rejestrów jest jednak znacznie ograniczona, tak więc są one używane zależnie od potrzeb. Nie mamy bezpośredniej kontroli ani też nie możemy dostrzec z poziomu naszych programów żadnych dowodów na to, że rejestry w ogóle istnieją (w C i C++ można kompilatorowi zasugerować przydział w rejestrze).

2. **Stos**. Jest on umieszczony w obszarze pamięci RAM (ang. *random-access memory*), ale jest także bezpośrednio obsługiwany przez procesor poprzez tzw. *wskaźnik stosu*. Wskaźnik stosu jest przesuwany w dół w celu zajęcia nowego obszaru pamięci, lub w górę — w celu jego zwolnienia. Jest to niezwykle szybki i efektywny sposób przydzielania pamięci — drugi po rejestrach. Kompilator Javy musi znać dokładny rozmiar oraz czas życia wszystkich danych przechowywanych na stosie podczas tworzenia programu, z uwagi na to, że musi wygenerować kod odpowiedzialny za przesunięcie wskaźnika stosu. Te wymogi ograniczają elastyczność programu, więc choć część danych jest przydzielana w pamięci stosu — w szczególności są to referencje do obiektów — to same obiekty nie są tam umieszczane.

3. **Sterta**. Jest to fragment pamięci ogólnego zastosowania (również w obrębie pamięci RAM), w którym przechowywane są wszystkie obiekty Javy. Pocieszające jest to, że — w przeciwieństwie do stosu — kompilator nie musi znać niezbędnego do zaalokowania rozmiaru pamięci ani też wiedzieć, jak długo dane będą zajmowały obszar na stercie. Zatem używając tego rodzaju magazynu, zyskujemy sporą elastyczność. Tam, gdzie potrzebny jest obiekt, piszemy po prostu kod tworzący obiekt za pomocą operatora `new`, a w momencie wykonania tego kodu obiektowi zostanie przydzielone miejsce w pamięci sterty. Oczywiście, istnieje cena, którą należy za nią zapłacić: przydzielenie pamięci sterty zajmuje więcej czasu niż w przypadku stosu (tzn. gdyby w ogóle dało się utworzyć obiekt w Javie na stosie, jak jest to możliwe w C++).

4. **Obszar stałych**. Wartości stałe są często umieszczane bezpośrednio w kodzie programu, co zabezpiecza je przed jakimikolwiek zmianami. Czasami same stałe są wydzielane z kodu, przez co mogą być ewentualnie umieszczone w pamięci tylko do odczytu (ROM)[2].

[2] Przykładem mogą być pule ciągów znaków: wszystkie literały łańcuchowe i inne stałe łańcuchowe mogą być wyodrębnione przez kompilator z kodu programu i przeniesione do specjalnego obszaru stałych.

5. **Obszar spoza RAM**. Jeżeli dane są zamieszczone poza programem, to mogą istnieć również wtedy, kiedy program nie jest uruchomiony, a więc są poza jego kontrolą. Dwa zasadnicze przykłady to *strumieniowanie obiektów*, podczas którego obiekty są przekształcane w strumień bajtów zazwyczaj po to, aby mogły być przesłane na inną maszynę, oraz *obiekty trwałe* umieszczane na dysku, a więc zachowujące swój stan nawet wtedy, gdy program zakończy działanie. Oba te sposoby zmieniają obiekty w coś, co może istnieć na innym medium, i dodatkowo, w razie potrzeby, może być ponownie wskrzeszone w postaci stałego obiektu umiejscowionego w pamięci RAM. Obecnie Java obsługuje tzw. *trwałość lekką* (ang. *lightweight persistence*); z kolei mechanizmy takie jak JDBC i Hibernate udostępniają daleko większe możliwości utrwalania i odzyskiwania obiektów; rolę pamięci trwałej dla obiektów pełni wtedy baza danych.

Przypadek specjalny: typy podstawowe

Istnieje grupa typów, które są traktowane w sposób szczególny. Można o nich myśleć jak o typach „podstawowych". Przyczyną takiego specjalnego traktowania jest fakt, że stworzenie obiektu poprzez zastosowanie operatora `new` — zwłaszcza obiektu małego, będącego prostą zmienną — nie jest zbyt wydajne, ponieważ operator ten przydziela pamięć na stercie. W przypadku omawianych typów Java zdaje się na wcześniejsze rozwiązanie, przejmując je z C i C++. Oznacza to, że zamiast tworzyć zmienną poprzez `new`, tworzy się zmienną „automatyczną", która *nie jest referencją*. Zmienna ta przechowuje wartość i jest umieszczana na stosie, co jest znacznie wydajniejsze.

Specyfikacja Javy określa dokładnie rozmiar każdego z typów podstawowych. Rozmiar ten jest niezmienny i niezależny od architektury sprzętowej, w przeciwieństwie do większości innych języków. Ta niezmienność rozmiaru jest jedną z przyczyn większej przenośności programów pisanych w Javie.

Nazwa typu	Rozmiar	Wartość minimalna	Wartość maksymalna	Typ obiektowy
`boolean`	—	—	—	`Boolean`
`char`	16 bitów	Unicode 0	Unicode $2^{16}-1$	`Character`
`byte`	8 bitów	-128	$+127$	`Byte`
`short`	16 bitów	-2^{15}	$+2^{15}-1$	`Short`
`int`	32 bity	-2^{31}	$+2^{31}-1$	`Integer`
`long`	64 bity	-2^{63}	$+2^{63}-1$	`Long`
`float`	32 bity	IEEE754	IEEE754	`Float`
`double`	64 bity	IEEE754	IEEE754	`Double`
`void`	—	—	—	`Void`

Wszystkie typy numeryczne są typami ze znakiem (ang. *signed*).

Rozmiar typu `boolean` nie jest jawnie określony; to taki typ, który w zbiorze dopuszczalnych wartości ma dwa literały: `true` (prawda) i `false` (fałsz).

Klasy „opakowujące" odpowiadające podstawowym typom danych pozwalają na tworzenie na stercie obiektów reprezentujących te typy, na przykład:

```
char c = 'x';
Character ch = new Character(c);
```

bądź równie dobrze:

```
Character ch = new Character('x');
```

Java SE5 przewiduje mechanizm *automatycznego pakowania* (ang. *autoboxing*) wartości typów podstawowych w obiektach, pozwalającego na konwersję wartości typu podstawowego na postać obiektu:

```
Character ch = 'x';
```

i odwrotnie:

```
char c = ch;
```

Powód, dla którego należy tak właśnie to robić, przedstawię później.

Liczby wysokiej precyzji

Język Java ma dwie klasy pozwalające na dokonywanie obliczeń arytmetycznych wysokiej precyzji: `BigInteger` oraz `BigDecimal`. Pomimo tego, że prawie pasują do tej samej kategorii co klasy „opakowujące", żadna z nich nie posiada odpowiednika w typach podstawowych.

Obie klasy posiadają metody, które pozwalają na operacje podobne do tych przewidzianych dla typów podstawowych. Zatem z `BigInteger` i `BigDecimal` można zrobić wszystko, co było możliwe z `int` i `float`, wystarczy jedynie użyć metod w miejsce operatorów. Jest to trochę bardziej skomplikowane, a więc i działanie będzie wolniejsze — zamieniamy prędkość działania na dokładność.

`BigInteger` reprezentuje typ całkowity dowolnej precyzji. Oznacza to, że można wiernie reprezentować zmienne całkowite dowolnych rozmiarów bez utraty jakichkolwiek informacji podczas obliczeń.

`BigDecimal` jest przeznaczona dla liczb stałoprzecinkowych dowolnej precyzji; klasy tej można przykładowo użyć w celu wykonywania dokładnych rachunków finansowych.

Szczegóły dotyczące konstruktorów i metod, które można stosować w tych dwu klasach, znajdziesz w dokumentacji.

Tablice w Javie

Praktycznie wszystkie języki programowania udostępniają tablice. Użycie tablic w C czy C++ okazuje się być niezbyt bezpieczne z uwagi na to, że w tych językach były one jedynie obszarami pamięci. Jeżeli program odwoływał się do tablicy poza przydzielonym jej obszarem albo próbował użyć pamięci przed jej inicjalizacją (częsty błąd programistów), otrzymywano zupełnie nieprzewidziane wyniki.

Jednym z głównych celów Javy jest bezpieczeństwo, stąd wiele problemów, które dotknęły programistów C i C++, tutaj nie występuje. Tablica w Javie gwarantuje bowiem, że zostanie zainicjowana, i nie może być dostępna poza swoją rozpiętością. Sprawdzanie zakresu powoduje drobne obciążenie poprzez dodatkową pamięć dla każdej tablicy, podobnie jest z kosztem sprawdzania indeksu podczas pracy programu, ale zyskana w rezultacie niezawodność oraz lepsza produktywność są tego warte (a do tego Java potrafi niekiedy zoptymalizować odwołania do tablic).

Kiedy tworzymy tablicę obiektów, to tak naprawdę tworzymy tablicę referencji do obiektów, z których każda jest automatycznie ustawiana na wyróżnioną wartość pustą oznaczoną słowem kluczowym null. Gdy kompilator Javy widzi taką referencję, rozpoznaje, że wcale nie wskazuje ona obiektu. Przed użyciem dowolnej referencji należy przypisać do niej jakiś obiekt, a jeżeli nadal próbowałbyś stosować puste referencje, podczas uruchomienia pojawi się komunikat. W ten sposób w Javie zapobiega się typowym błędom odwołań do elementów tablic.

Można również utworzyć tablicę elementów typów podstawowych. Kompilator zapewnia inicjalizację poprzez wyzerowanie obszaru przydzielonego dla takiej tablicy.

O tablicach napiszę szczegółowo w jednym z dalszych rozdziałów.

Nigdy nie ma potrzeby niszczenia obiektu

W większości języków programowania zajmowanie się czasem życia zmiennych stanowi istotną część całego wysiłku programisty. Jak długo zmienna przetrwa? Jeżeli powinno się ją zniszczyć, to kiedy? Nieład związany z cyklem życia zmiennych może prowadzić do wielu błędów, dlatego też w tym podrozdziale zobaczysz, jak bardzo Java upraszcza to poprzez czyszczenie pamięci za programistę.

Zasięg

Większość języków proceduralnych posługuje się pojęciem *zasięgu* (ang. *scope*). Określa ono zarówno widoczność, jak i czas życia zmiennych zdefiniowanych w ramach zasięgu. W C, C++ i Javie zasięg jest określony poprzez położenie nawiasów klamrowych {}. Tak więc na przykład:

```
{
  int x = 12;
  // tylko x jest dostępny
  {
    int q = 96;
    // dostępne są obie zmienne x i q
  }
  // tylko x jest dostępny
  // q jest poza zasięgiem ("out of scope")
}
```

Zmienna zdefiniowana w danym zasięgu jest dostępna tylko do miejsca jego zakończenia.

Dowolny tekst umieszczony za dwoma ukośnikami (//) jest, aż do końca wiersza, traktowany jako komentarz.

Stosowane powyżej wcięcia zapewniają lepszą czytelność kodu Javy, natomiast dodatkowe znaki odstępu, tabulatory oraz przejścia do nowego wiersza nie mają wpływu na program wynikowy.

Zauważ, że poniższa konstrukcja, choć poprawna w C i C++, w Javie jest *niedozwolona*:

```
{
  int x = 12;
  {
    int x = 96; // niedozwolone
  }
}
```

Kompilator poinformuje, że zmienna x została już wcześniej zdefiniowana, a zatem języki C i C++ mają możliwość „ukrywania" zmiennych, która nie jest dostępna w Javie. Jej projektanci uznali, że prowadzi to do nieczytelności kodu.

Zasięg obiektów

Czas życia obiektów Javy różni się od czasu życia zmiennych typów podstawowych. Kiedy w Javie tworzymy obiekt poprzez operator new, to obiekt istnieje również poza swoim zasięgiem. Zatem jeżeli zapiszemy:

```
{
  String s = new String("łańcuch");
} // koniec zasięgu
```

to referencja s przepadnie wraz z końcem zasięgu. Obiekt typu String, na który wskazywała referencja s, będzie natomiast wciąż zajmował pamięć. W powyższym fragmencie kodu nie ma żadnego sposobu uzyskania dostępu do tego obiektu, ponieważ jedyna referencja jest już poza zasięgiem. W dalszej części książki zobaczysz, jak referencje do obiektów mogą być przekazywane i kopiowane w trakcie działania programu.

Dzieje się tak, ponieważ obiekty utworzone za pomocą new są przechowywane tak długo, jak długo są potrzebne, dzięki czemu wszystkie skomplikowane problemy programowania w C++ tutaj po prostu znikają. W C++ nie tylko należy dbać o to, by obiekty były dostępne wtedy, gdy są potrzebne, wymagane jest również zniszczenie obiektu, kiedy nie jest już używany.

W związku z tym nasuwa się interesujące pytanie: jeżeli w Javie obiekty są ciągle przechowywane, to co chroni nas przed przepełnieniem całej pamięci i zablokowaniem działania programu? Taki problem powstałby w przypadku C++. Tu mamy do czynienia z odrobiną magii. Java posiada bowiem *odśmiecacz pamięci* (ang. *garbage collector*), który sprawdza wszystkie obiekty utworzone poprzez operator new i wykrywa te, do których nie ma aktualnych referencji. Następnie zwalnia pamięć przydzieloną tym obiektom, dzięki czemu może być ona wykorzystana przez kolejne obiekty. Oznacza to, że programista nie musi się nigdy martwić o zwrot pamięci — zwyczajnie tworzy potrzebny mu obiekt, a kiedy go już nie potrzebuje, ginie on samoczynnie. Zatem pozbywamy się pewnego typu problemów, nazywanych również „wyciekaniem pamięci", powstających, kiedy programista zapomina o zwalnianiu pamięci.

Własne typy danych — słowo class

Jeżeli wszystko jest obiektem, to co wyznacza wygląd i zachowanie konkretnego obiektu? Innymi słowy, co stanowi o *typie* obiektu? Można by oczekiwać, że będzie temu służyło słowo kluczowe „type" i oczywiście miałoby to sens. Większość języków zorientowanych obiektowo używa jednak słowa kluczowego `class` w znaczeniu: „Określam właśnie, jak prezentuje się nowy typ obiektu". Po słowie `class` (które jest na tyle powszechne, że nie będzie już dalej wyróżniane w zapisie) następuje nazwa nowego typu, na przykład:

```
class NazwaTypu { /* ciało klasy zamieszczamy tutaj */ }
```

W ten sposób stworzony został nowy typ, choć ciało klasy zawiera jedynie komentarz (gwiazdki i ukośniki oraz wszystko pomiędzy omówię trochę później w tym rozdziale), a więc niezbyt wiele można z tym zrobić. Jednak można już tworzyć obiekty tego typu poprzez zastosowanie operatora `new`:

```
NazwaTypu a = new NazwaTypu();
```

Tak naprawdę nie można jej niczego kazać (czyli nie da się jej przesłać żadnych komunikatów), zanim nie zdefiniujemy jakichś metod.

Pola i metody

Kiedy definiujemy klasę (a wszystko, co robi się, programując w Javie, to definiowanie klas, tworzenie z nich obiektów oraz przesyłanie do nich komunikatów), możemy w niej zamieścić dwa rodzaje elementów: *pola* (zmienne nazywane czasami *danymi składowymi*) i *metody* (zwane też *funkcjami składowymi*). Pola to obiekty dowolnych typów dostępne za pośrednictwem referencji; mogą też być wartościami typów podstawowych. Jeżeli mamy do czynienia z referencją do obiektu, to należy taką zmienną zainicjować, aby powiązać ją z rzeczywistym obiektem (poprzez operator `new`, jak to było pokazane wcześniej).

Każdy z obiektów utrzymuje własny obszar pamięci dla swoich pól; zmienne składowe nie są współdzielone pomiędzy obiektami. Poniżej mamy przykład klasy z kilkoma składowymi:

```
class TylkoDane {
  int i;
  double d;
  boolean b;
}
```

Klasa ta nie robi zupełnie nic, ale można stworzyć jej obiekt:

```
TylkoDane dane = new TylkoDane();
```

Można też przypisać wartości polom, jednak wcześniej powinieneś się dowiedzieć, jak odwoływać się do składowych obiektu. Otóż można tego dokonać poprzez podanie nazwy referencji do obiektu, kropki oraz nazwy składowej z wnętrza obiektu:

```
referencjaObiektu.składowa
```

Spójrzmy na przykłady:

```
dane.i = 47;
dane.d = 1.1;
dane.b = false;
```

Jest także możliwe, że obiekt będzie zawierał inne obiekty zawierające z kolei dane, które chcielibyśmy móc modyfikować. Wtedy po prostu należy nadal „zespalać wszystko kropkami", np.:

```
mójSamolot.lewyZbiornik.pojemność = 100;
```

Klasa `TylkoDane` nie może zdziałać zbyt wiele poza przechowywaniem danych z uwagi na to, że nie zawiera funkcji składowych (metod). Aby zrozumieć, jak takie metody funkcjonują, należy wcześniej zapoznać się z argumentami oraz wartościami zwracanymi, które pokrótce opiszę.

Wartości domyślne dla składowych typów podstawowych

W przypadku gdy składową klasy jest zmienna typu podstawowego, to zawsze, jeśli nie zostanie zainicjowana jawnie, otrzymuje wartość domyślną:

Nazwa typu	Wartość domyślna
`boolean`	`false`
`char`	`'\u0000'`
`byte`	`(byte)0`
`short`	`(short)0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`

Wartości domyślne są gwarantowane, gdy zmienne są stosowane *jako składowe klasy*. Dzięki temu mamy pewność, że składowe typów podstawowych będą zawsze zainicjowane (w C++ tego nie ma), redukując tym samym kolejne źródło błędów. Mimo to tak zainicjowane wartości mogą być niewłaściwe lub nawet nieodpowiednie w konkretnej implementacji programu, dlatego zawsze najlepiej zainicjować zmienne samodzielnie.

Powyższe przypisanie wartości domyślnych nie dotyczy *zmiennych lokalnych* — tych, które nie są polami w klasie. Tak więc, jeśli wewnątrz definicji funkcji mamy:

```
int x;
```

to `x` uzyska wartość nieokreśloną (podobnie jak w C czy C++); nie będzie automatycznie inicjowany na wartość zero. To programista odpowiada za przypisanie właściwej wartości, zanim spróbuje użyć zmiennej `x`. Java przewyższa C++ nawet wtedy, gdy programista o tym zapomni: podczas kompilacji dostaniemy informację o błędzie polegającym na tym, że zmienna może nie być zainicjowana (wiele kompilatorów C++ również informuje o niezainicjowanych zmiennych, ale w Javie są to już sytuacje uznawane za błędy).

Metody, argumenty i wartości zwracane

Do tej pory w celu określenia nazwanego podprogramu używaliśmy terminu *funkcja*. Określeniem, które w Javie jest znacznie powszechniej stosowane, jest *metoda* — „sposób na wykonanie czegoś". Jeśli chcesz, to oczywiście możesz nadal myśleć w kategoriach funkcji. Różnica występuje jedynie na poziomie składniowym, jednak, zgodnie z popularną konwencją, w książce będę stosować termin „metoda".

Metody w Javie określają komunikaty, które może otrzymać dany obiekt. Zasadniczymi częściami każdej metody są: nazwa, pobierane argumenty, typ zwracany oraz ciało. Oto podstawowa postać:

```
TypZwracany nazwaMetody( /* lista argumentów */ ) {
  /* ciało metody */
}
```

Typ zwracany to typ wartości, która jest zwracana z metody po jej wywołaniu. Lista argumentów zawiera typy oraz nazwy informacji, które chcemy przekazać do metody. Nazwa oraz lista argumentów metody (zwane też *sygnaturą metody*) razem pozwalają w jednoznaczny sposób identyfikować daną metodę.

Metody w Javie mogą być tworzone wyłącznie jako części składowe klasy. Mogą być wywoływane jedynie na rzecz obiektu[3], a taki obiekt musi być do tego upoważniony. Jeżeli będziemy usiłowali wywołać niewłaściwą metodę obiektu, to podczas kompilacji otrzymamy komunikat o błędzie. Wywołanie metody polega na podaniu nazwy obiektu, kropki, nazwy metody oraz jej listy argumentów, np.:

```
nazwaObiektu.nazwaMetody(arg1, arg2, arg3);
```

Przypuśćmy, że mamy metodę o nazwie `f()`, która nie pobiera żadnych argumentów oraz zwraca wartość typu całkowitego `int`. Dalej, jeśli mamy obiekt o nazwie `a`, dla którego można wywołać `f()`, to można wykonać działanie:

```
int x = a.f();
```

Typ wartości zwracanej musi być zgodny z typem zmiennej `x`.

To postępowanie podczas wywoływania metod jest powszechnie określane jako *wysłanie komunikatu do obiektu*. W powyższym przykładzie komunikatem jest `f()` a obiektem `a`. Samo programowanie obiektowe jest także często rozumiane jako po prostu „komunikacja z obiektami".

Lista argumentów

Lista argumentów metody określa, jakie informacje możemy przekazać do jej wnętrza. Jak można się domyślić, informacja ta — jak wszystko inne w Javie — przybiera postać obiektu. Tak więc w liście argumentów należy określić typy przekazywanych obiektów

[3] Metody statyczne, o których więcej dowiesz się wkrótce, mogą być wywoływane na rzecz klasy bez konieczności istnienia żadnego jej obiektu.

oraz nazwy każdego z nich. Podobnie — jak w dowolnej sytuacji w Javie — kiedy chcemy przekazać obiekty, w rzeczywistości przekazujemy ich referencje[4]. Typ referencji powinien być oczywiście odpowiedni. Jeśli argument ma być przykładowo typu `String`, to koniecznie musimy przekazać łańcuch tekstowy.

Rozważmy metodę, która właśnie jako argument pobiera obiekt typu `String`. Oto definicja, którą trzeba by zamieścić w obrębie definicji klasy:

```
int wielkość(String s) {
  return s.length() * 2;
}
```

Powyższa metoda zwraca wartość określającą, ile bajtów jest niezbędnych do przechowywania w pamięci konkretnego łańcucha tekstowego (każdy znak typu `char` w łańcuchu tekstowym to 16 bitów lub inaczej 2 bajty, gdyż mamy do czynienia z zapisem w standardzie Unicode). Argument jest zatem typu `String`, a jego nazwa to `s`. Po przekazaniu `s` do metody można go traktować jak zwyczajny obiekt (można przesyłać do niego komunikaty). Powyżej wywoływana jest metoda `length()`, będąca jedną z metod klasy `String` — zwraca ona liczbę znaków łańcucha tekstowego.

Użyte tu zostało również słowo kluczowe `return`, które odpowiada za dwie rzeczy. Po pierwsze, oznacza „Opuść metodę, bo się zakończyła", a po drugie, jeżeli metoda ma zwracać wartość, to jest ona umieszczana właśnie tuż po słowie `return`. W tym przypadku wartość powstaje po wykonaniu wyrażenia `s.length() * 2`.

Oczywiście można zwrócić dowolny typ, jaki sobie zażyczymy, ale kiedy nie chcemy zwrócić nic, wystarczy wskazać, że metoda zwraca typ `void`. Oto kilka przykładów:

```
boolean znacznik() { return true; }
float podstawaLogarytmuNaturalnego() { return 2.718f; }
void nic() { return; }
void nic2() {}
```

Jeśli typem zwracanym jest `void`, to słowo kluczowe `return` stosuje się jedynie w celu określenia punktu wyjścia z metody i dlatego jego użycie na końcu metody nie jest wymagane. Wyjście z metody może nastąpić w dowolnym jej miejscu, ale jeśli wskażemy inny typ niż `void`, to kompilator wymusi (poprzez komunikat o błędzie) podanie wartości właściwego typu.

Może się wydawać, że program jest po prostu zbiorem obiektów zawierających metody, które pobierają jako argumenty inne obiekty i przesyłają do nich komunikaty. Jest w tym stwierdzeniu rzeczywiście sporo prawdy — w następnym rozdziale dowiesz się dokładnie, jak pisać ciała metod. W tym rozdziale przesyłanie komunikatów powinno Ci wystarczyć.

[4] Z wyjątkiem wcześniej wspomnianych „specjalnych" typów danych: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` oraz `double`. Zazwyczaj, mimo że przekazujemy obiekty, naprawdę znaczy to, że przekazujemy referencje do nich.

Tworzenie programu w Javie

Istnieje kilka innych zagadnień, które powinieneś zrozumieć, zanim napiszesz swój pierwszy program w Javie.

Widoczność nazw

Właściwa kontrola używanych w programach nazw jest zagadnieniem dotyczącym każdego języka programowania. Jeżeli użyjemy danej nazwy w jednym module programu i inny programista zastosuje taką samą nazwę w innym, to powstaje pytanie, jak odróżnić jedną nazwę od drugiej i zapobiec „kolizji" obu nazw. W języku C jest to problem szczególny, ponieważ program często jest tam wręcz niemożliwym do opanowania morzem nazw. W C++ klasy (na których opierają się klasy Javy) zawierają funkcje już w swym wnętrzu, a więc nie mogą się one pokryć z funkcjami innych klas. C++ wciąż jednak zezwala na używanie globalnych zmiennych i funkcji, toteż konflikty nazw są nadal możliwe. Aby rozwiązać ten problem, w C++ wprowadzono *przestrzenie nazw* (ang. *namespaces*) poprzez dodatkowe słowo kluczowe.

W przypadku Javy wszystkie kłopoty udało się ominąć dzięki nowemu rozwiązaniu. Aby uzyskać jednoznaczne nazwy bibliotek, należy posługiwać się nazwą domeny internetowej. Twórcy Javy proponują stosowanie odwróconej nazwy domeny internetowej, gdyż dzięki temu mamy pewność niepowtarzalności. Jako że moja domena to `MindView.net`, to własną bibliotekę usługową `foibles` powinienem nazwać `net.mindview.utility.foibles`. Po odwróceniu nazwy domeny kropki można rozumieć jako znaki reprezentujące przejście do podkatalogu.

W Java 1.0 oraz 1.1 domeny główne *com*, *edu*, *org*, *net* itp. były z zasady zamieniane na duże litery, czyli nazwa biblioteki miałaby postać: `NET.mindview.utility.foibles`. Później okazało się, że może to powodować pewne problemy, toteż obecnie nazwy pakietów są pisane małymi literami.

Mechanizm ten oznacza, że wszystkie pliki automatycznie znajdują się w swoich własnych przestrzeniach nazw oraz że każda klasa w pliku musi posiadać unikatowy identyfikator. Dzięki temu nie ma potrzeby poznawania wyszukanych cech języka, aby uporać się z tym problemem — język dba o to za nas.

Wykorzystanie innych komponentów

Zawsze, kiedy chcemy użyć klasy już zdefiniowanej, kompilator powinien wiedzieć, jak taką klasę zlokalizować. Oczywiście klasa może istnieć w tym samym pliku źródłowym, z którego została wywołana. W takim przypadku po prostu wystarczy jej użyć — nawet kiedy klasa ta nie jest jeszcze w pełni zdefiniowana (kłopoty z „wcześniejszym odwołaniem" zostały w Javie wyeliminowane, a więc nie ma potrzeby się nimi zajmować).

Co z klasami, które są zamieszczone w innych plikach? Można przypuszczać, że kompilator powinien być na tyle sprytny, żeby po prostu pójść i poszukać, ale nie jest to takie proste. Wyobraźmy sobie, że chcemy wykorzystać klasę o określonej nazwie, ale istnieje więcej niż jedna definicja klasy (przypuszczalnie są to odmienne definicje). Można

też sobie wyobrazić, że piszemy program i w trakcie pracy dodajemy do biblioteki nową klasę, której nazwa powoduje konflikt z jedną z wcześniejszych klas.

Aby zaradzić powyższym problemom, należałoby wyeliminować wszystkie potencjalne dwuznaczności. Robi się to poprzez dokładne wskazanie kompilatorowi Javy, której klasy potrzebujemy, stosując słowo kluczowe `import`. Słowo to nakazuje kompilatorowi wprowadzić nowy *pakiet*, który jest biblioteką klas (w innych językach biblioteki oprócz klas mogłyby składać się z funkcji i danych, ale w Javie należy pamiętać, że wszystko piszemy wewnątrz klasy).

Przez większość czasu zapewne będziesz używał istniejących komponentów ze standardowej biblioteki Javy dostarczanej wraz z kompilatorem. W takim przypadku nie musisz się martwić o długie, odwrócone nazwy domen, np. aby poinformować kompilator, że chcemy zastosować klasę o nazwie `ArrayList`, stosuje się zapis:

```
import java.util.ArrayList;
```

Pakiet `util` zawiera wiele klas, a w związku z tym można stosować kilka z nich bez wyraźnego określania. Można to łatwo uzyskać poprzez wpisanie symbolu wieloznacznego *:

```
import java.util.*;
```

Jest to sposób bardziej powszechny niż wskazywanie każdej klasy z osobna, kiedy chcemy włączyć kolekcję klas.

Słowo kluczowe static

Zazwyczaj, kiedy tworzymy klasę, opisujemy, jak będą wyglądać i zachowywać się jej obiekty. Właściwie niczego nie możemy uzyskać, zanim dzięki operatorowi `new` nie zostanie utworzony obiekt danej klasy. Od tej pory przydzielona zostaje pamięć dla danych oraz stają się dostępne metody.

Są jednak dwie sytuacje, w których powyższy sposób nie wystarcza. Pierwszy przypadek to sytuacja, kiedy chcemy mieć tylko jeden obszar pamięci dla konkretnych danych, niezależnie od tego, jak wiele obiektów zostanie stworzonych lub nawet kiedy żaden obiekt nie będzie utworzony. Inna sytuacja to potrzeba posiadania metody, która nie byłaby związana z żadnym konkretnym obiektem danej klasy, a więc chcielibyśmy zyskać metodę, którą można by wywołać nawet wtedy, kiedy nie byłby utworzony żaden obiekt.

Oba efekty można osiągnąć poprzez zastosowanie słowa kluczowego `static`. Jeżeli mówimy, że coś jest statyczne, oznacza to, że składowa lub metoda nie jest związana z żadnym konkretnym egzemplarzem klasy. Zatem nawet jeżeli nigdy nie utworzymy obiektu danej klasy, to można wywołać jej metodę statyczną lub pozyskać dostęp do statycznych składowych. W przypadku zwykłych, niestatycznych składowych i metod odwołanie do pola albo wywołanie metody wymaga utworzenia i użycia obiektu — niestatyczne składowe i metody muszą być zawsze odnoszone do konkretnego egzemplarza klasy[5].

[5] Oczywiste jest, iż mimo że metody statyczne nie wymagają stworzenia obiektu przed ich użyciem, to nie mogą *bezpośrednio* uzyskać dostępu do składowych i metod niestatycznych poprzez zwykłe zawołanie składowych bez odwołania do nazwanego obiektu (wszak składowe i metody niestatyczne muszą być powiązane z konkretnym obiektem).

Niektóre języki obiektowe stosują określenie *zmiennej klasowej* i *metody klasowej*, co oznacza, że składowa lub metoda reprezentuje stan lub zachowanie klasy jako takiej, a nie żadnego konkretnego egzemplarza klasy. Czasami literatura dotycząca Javy również używa tych określeń.

Aby utworzyć składową lub metodę statyczną, wystarczy umieścić słowo kluczowe `static` przed samą definicją. W poniższym przykładzie tworzymy i inicjujemy statyczną składową:

```
class StaticTest {
    static int i = 47;
}
```

Teraz, jeżeli nawet utworzymy dwa obiekty typu `StaticTest`, to wciąż będzie istnieć jeden obszar pamięci dla składowej `StaticTest.i`. Oba obiekty będą współdzielić składową `i`. Rozważmy:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

Odwołania `st1.i` oraz `st2.i` dają tę samą wartość, równą 47, z tego względu, że odnoszą się do tego samego obszaru pamięci.

Istnieją dwa sposoby dostępu do zmiennej statycznej. Jak sygnalizowałem wcześniej, można zrobić to poprzez obiekt, pisząc przykładowo `st2.i`. Można także odwołać się bezpośrednio poprzez nazwę klasy — nie można tego wykonać w przypadku składowych nie będących statycznymi.

```
StaticTest.i++;
```

Operator `++` inkrementuje zmienną (zwiększa o jeden). Tak więc teraz `st1.i` oraz `st2.i` będą miały wartość 48.

Odwołanie do składowej statycznej najlepiej wykonywać za pośrednictwem nazwy klasy (a nie nazwy obiektu). W ten sposób podkreśla się statyczność zmiennej, a do tego niekiedy można takie odwołanie zoptymalizować.

Analogicznie jest w przypadku statycznych metod. Można odwoływać się do metod statycznych albo poprzez obiekt, jak w przypadku dowolnych metod, albo stosując dodatkową składnię `NazwaKlasy.metoda()`. Metody statyczne definiuje się podobnie:

```
class StaticFun {
  static void incr() { StaticTest.i++; }
}
```

Metoda `incr()` z klasy `StaticFun` inkrementuje zmienną statyczną `i`, wykorzystując do tego celu operator `++`. Można ją wywołać w typowy sposób poprzez obiekt:

```
StaticFun sf = new StaticFun();
sf.incr();
```

lub, ponieważ jest to metoda statyczna, bezpośrednio poprzez klasę:

```
StaticFun.incr();
```

Choć zastosowanie słowa `static` do zmiennej całkowicie zmienia sposób jej tworzenia (jedna dla klasy w przeciwieństwie do jednej dla obiektu w przypadku zwykłych zmiennych), to w przypadku metod nie jest aż tak źle. Ważnym zastosowaniem metod statycznych jest pozyskanie możliwości wywołania takiej metody bez konieczności tworzenia obiektu. Jak wkrótce się przekonasz, jest to niezbędne w przypadku definicji metody `main()`, której wywołanie rozpoczyna działanie każdej aplikacji.

Twój pierwszy program w Javie

Wreszcie napiszemy jakiś program. Zaczniemy od wypisania łańcucha tekstowego oraz daty poprzez użycie klasy `Date` ze standardowej biblioteki Javy.

```
// HelloDate.java
import java.util.*;

public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Witaj, dzisiaj jest: ");
    System.out.println(new Date());
  }
}
```

Na początku każdego programu należy zamieścić wyrażenie `import`, aby wprowadzić dodatkowe klasy, które będą potrzebne w kodzie zamieszczonym w danym pliku. Zauważ, że określiłem to słowem „dodatkowe", a to z tego względu, że pewna biblioteka klas jest włączana automatycznie do każdego pliku Javy — jest nią `java.lang`. Uruchom przeglądarkę internetową i zajrzyj do dokumentacji dostarczonej przez firmę Sun (jeśli nie ściągnąłeś jej ze strony *java.sun.com* lub nie zainstalowałeś inaczej, to zrób to teraz[6]; pamiętaj, że dokumentacja jest udostępniana w osobnym pakiecie, dystrybuowanym niezależnie od JDK). Jeżeli przyjrzysz się spisowi pakietów, to zobaczysz wszystkie biblioteki klas dostarczane z Javą. Wybierz `java.lang`. Spowoduje to pojawienie się listy klas zawartych w tej bibliotece. Wszystkie te klasy są dostępne w sposób automatyczny, ponieważ `java.lang` jest włączany pośrednio do każdego pliku z kodem Javy. W pakiecie `java.lang` nie ma klasy `Date`, co znaczy, iż musi być ona dołączana z innej biblioteki. Jeżeli nie znamy nazwy biblioteki, w której zamieszczona jest interesująca nas klasa, lub chcemy przejrzeć wszystkie klasy, to wystarczy z dokumentacji wybrać odnośnik „Tree". Teraz można już odnaleźć każdą klasę, którą dostajemy wraz z Javą. Dalej możesz w przeglądarce użyć opcji „Find", aby odszukać klasę `Date`. Po tym zorientujesz się, że jest ona wypisana jako `java.util.Date`, co pozwala przypuszczać, że jest zamieszczona w bibliotece `util`, oraz że trzeba zaimportować `java.util.*`, aby móc jej używać.

Jeżeli powrócisz do początku dokumentacji, wybierz też `java.lang`, a następnie `System`. Zobaczysz, że klasa `System` zawiera kilkanaście pól, i jeżeli wybierzesz pole `out`, przekonasz się, że jest to obiekt statyczny klasy `PrintStream`. Obiekt `out` jest tam zawsze, nie trzeba go tworzyć, a wystarczy po prostu użyć. To, co można dzięki niemu zrobić, zależy od jego typu, czyli od klasy `PrintStream`. Na szczęście nazwa tej klasy również widnieje jako odnośnik, zatem jeżeli go klikniesz, to wyświetli się lista jego metod. Jest ich zaledwie kilka, ale opiszę je dopiero później. Póki co interesuje nas jedynie metoda `println()`, a jej

[6] Kompilator i dokumentacja Javy zmieniają się zbyt często, więc najlepiej zaopatrywać się w nie bezpośrednio w witrynie firmy Sun. Wtedy można mieć pewność, że dysponuje się najnowszą dostępną wersją.

użycie oznacza: „Wypisz na konsolę to, co podałem, i dołącz znak przejścia do nowego wiersza". Zatem w dowolnym programie napisanym w Javie można wpisać:

```
System.out.println("cokolwiek");
```

jeżeli tylko chcemy wypisać „cokolwiek" na konsolę.

Nazwa klasy publicznej powinna odpowiadać nazwie pliku. Tworząc niczależny program, taki jak ten, jedna z klas umieszczonych w pliku musi mieć taką samą nazwę jak nazwa pliku. (Jeśli takiej klasy nie będzie, kompilator wyświetli stosowny komunikat o błędzie.) Klasa ta musi zawierać metodę o nazwie `main()`, o następującej sygnaturze:

```
public static void main(String[] args) {
```

Słowo kluczowe `public` informuje o dostępności metody na zewnątrz (szczegóły poznasz w rozdziale „Kontrola dostępu"). Argumentem przekazywanym do metody jest tablica obiektów typu `String`. Argument `args`, mimo że nie jest używany w naszym programie, jest wymagany przez kompilator, gdyż przechowuje argumenty wywołania programu z wiersza poleceń.

Wiersz, w którym wypisujemy datę, jest dosyć ciekawy:

```
System.out.println(new Date());
```

Spójrz na argument: obiekt klasy `Date` jest tu tworzony po to, by przekazać wartość (automatycznie konwertowaną do postaci obiektu `String`) do metody `println()`. Gdy tylko polecenie zostanie wykonane, obiekt jest już niepotrzebny i może być w każdej chwili usunięty z pamięci przez mechanizm odśmiecania. Nie musimy w ogóle się tym zajmować.

Przeglądając dokumentację JDK pobraną z witryny *http://java.sun.com*, zauważysz z pewnością, że klasa `System` posiada również wiele innych metod pozwalających na generowanie ciekawych rezultatów (zresztą dostępność bogatych bibliotek standardowych to jedna z większych zalet Javy). Oto przykład:

```
//: object/ShowProperties.java

public class ShowProperties {
  public static void main(String[] args) {
    System.getProperties().list(System.out);
    System.out.println(System.getProperty("user.name"));
    System.out.println(
      System.getProperty("java.library.path"));
  }
} ///:~
```

Pierwszy wiersz powyższej metody `main()` wypisuje wszystkie „właściwości" systemu, w którym uruchomiono program, a więc udostępnia informacje o środowisku wykonania programu. Metoda `list()` przekazuje rezultat swojego działania do argumentu jej wywołania: `System.out`. Przekonasz się wkrótce, że dane można by przekazać również gdzie indziej, choćby do pliku. Można też wybierać interesujące właściwości środowiska — przykład pokazuje, jak dobrać się do nazwy użytkownika (właściwość `user.name`) i ścieżki do bibliotek Javy (właściwość `java.library.path`). Znaczenie nietypowych komentarzy na początku i końcu kodu wyjaśni się później.

Kompilacja i uruchomienie

Aby skompilować i uruchomić nasz program, podobnie jak każdy inny program z tej książki, trzeba najpierw zaopatrzyć się w środowisko programowania Javy. Jest wiele środowisk pochodzących od różnych firm, ale przyjmuję, że będziesz stosował JDK firmy Sun, które jest darmowe. W innym przypadku będziesz musiał przejrzeć dokumentację danego środowiska, aby ustalić, jak w nim kompilować i uruchamiać programy[7].

Połącz się z Internetem i odwiedź witrynę *http://java.sun.com*. Tam znajdziesz informacje oraz odnośniki, które opiszą Ci cały proces ściągania i instalacji pakietu JDK dla Twojego systemu operacyjnego.

Jeżeli już zainstalujesz środowisko JDK oraz poustawiasz ścieżki dostępu, tak aby mieć dostęp do programów *javac* i *java*, możesz ściągnąć i rozpakować kody przykładów do książki (znajdują się one na stronie *www.MindView.net*). Po rozpakowaniu uzyskasz odpowiednie podkatalogi dla każdego z rozdziałów książki. Wejdź do podkatalogu o nazwie *object* i wydaj polecenie:

```
javac HelloDate.java
```

Polecenie to nie powinno niczego wypisać w odpowiedzi. Jeśli jednak pojawi się jakiś rodzaj błędu, oznacza to, że JDK zostało zainstalowane nieprawidłowo i pozostaje Ci to zbadać. Jeżeli ponownie pojawi się znak zachęty, to możesz wpisać:

```
java HelloDate
```

dzięki czemu otrzymasz komunikat o aktualnej dacie.

Tym sposobem można kompilować i uruchamiać każdy program dołączony do książki. Zauważysz zapewne, że do kodów źródłowych przykładów przedstawionych w niniejszej książce, w katalogach odpowiadających poszczególnym rozdziałom, znajdują się pliki *build.xml*. Zawierają one instrukcje programu *Ant* umożliwiające automatyczne „budowanie" przykładów zamieszczonych w danym katalogu. Pliki budowy oraz program Ant zostały dokładniej opisane w dodatku publikowanym pod adresem *http://MindView.net/Books/BetterJava*; w skrócie, kiedy już zainstalujesz program *Ant* (można go pobrać ze strony *http://ant.apache.org*), aby skompilować i uruchomić wszystkie programy w danym katalogu, wystarczy wydać w wierszu poleceń polecenie ant. Jeśli nie dysponujesz programem Ant, możesz skompilować i uruchomić przykłady własnoręcznie, posługując się poleceniami javac oraz java.

[7] Popularną alternatywą dla Sun JDK jest kompilator „jikes" firmy IBM, który jest szybszy od swego odpowiednika — javac (ale przy kompilowaniu grupowym, za pomocą programu Ant, różnica jest minimalna). Pojawiły się też „wolne" (spod egidy open source) projekty kompilatorów, bibliotek i środowisk wykonawczych języka Java.

Komentarze oraz dokumentowanie kodu

Istnieją dwa rodzaje komentarzy. Pierwszy to tradycyjny komentarz języka C, który został odziedziczony przez C++. Rozpoczyna się od znaków /*, może obejmować wiele wierszy, a kończy się znakami */. Wielu programistów dodatkowo rozpoczyna każdy wiersz takiego komentarza znakiem *, toteż można to często zobaczyć w postaci:

```
/* To jest komentarz,
 * który rozciąga się na
 * wiele wierszy
 */
```

Należy jednak pamiętać, że wszystko, co znajduje się pomiędzy /* a */, jest ignorowane, a więc powyższy komentarz można zapisać tak:

```
/* To jest komentarz, który
rozciąga się na wiele wierszy */
```

Druga wersja komentarza pochodzi już z C++. Jest to komentarz jednowierszowy, rozpoczynający się od znaków // i kończący wraz z końcem wiersza. Ten typ komentarza jest dosyć wygodny i powszechnie stosowany z uwagi na prostotę użycia. Nie ma potrzeby poszukiwać klawisza /, a następnie * (wystarczy kliknąć jeden klawisz dwukrotnie), no i nie ma potrzeby zamknięcia komentarza. Tak więc często może się pojawić forma:

```
// to jest komentarz jednowierszowy
```

Dokumentacja w komentarzach

Projektanci języka Java nie uważali, że jedyną istotną rzeczą jest tworzenie kodu programów, ale również zatroszczyli się o jego dokumentowanie. Prawdopodobnie największe kłopoty z dokumentacją kodu sprawia jej przechowywanie. Jeżeli bowiem dokumentacja jest wydzielona z kodu, to zmiana opisu może być sporym problemem, gdy zmienimy kod. Rozwiązanie wydaje się proste: powiązać kod z dokumentacją. Najprostszym sposobem, aby to osiągnąć, jest umieszczenie całości w jednym pliku. Potrzebny będzie specjalny zapis komentarzy oraz dodatkowe narzędzie, które pozwoli wydobyć taki opis i zapisać go w formie bardziej użytecznej. Wszystko to oczywiście jest już gotowe.

Narzędzie, które wyciąga komentarze z kodu, nosi nazwę *Javadoc*. Program ten stosuje kilka mechanizmów z kompilatora Javy, aby odnaleźć specjalne znaczniki komentarzy wewnątrz kodu. Jednak wyciągane są nie tylko informacje zawarte w znacznikach, ale również nazwy klas i metod, które im towarzyszą. Tym sposobem możemy przy minimalnym nakładzie pracy wygenerować przyzwoitą dokumentację.

Wynikiem działania programu *Javadoc* jest plik w formacie HTML, który można obejrzeć w dowolnej przeglądarce internetowej. Narzędzie to stosuje się do pojedynczych plików z kodem Javy. Dzięki niemu powstał standard dokumentowania kodu, który jest na tyle prosty, że możemy spodziewać się uzyskać, a nawet żądać, dokumentacji ze wszystkimi bibliotekami Javy.

Dodatkowo można także tworzyć specjalne procedury obsługi komentarzy, nazywane *docletami*, pozwalające na wykonywanie specjalnych operacji (takich jak zapis danych w innym formacie) na informacjach przetwarzanych przez narzędzie *Javadoc*. Zagadnienia te zostały opisane w osobno publikowanym suplemencie, dostępnym pod adresem *http://MindView.net/Books/BetterJava*.

Dalsza część rozdziału zawiera jedynie wprowadzenie i przegląd podstawowych możliwości programu *Javadoc*. Jego szczegółowy opis jest dołączony do dokumentacji JDK. Po pobraniu i rozpakowaniu dokumentacji poszukaj odpowiedniego pliku w podkatalogu „tooldocs" (lub skorzystaj z odnośnika „tooldocs").

Składnia

Komentarze *Javadoc* występują wyłącznie po rozpoczynających znakach `/**` i kończą się, jak w przypadku zwykłych komentarzy, znakami `*/`. Istnieją dwa podstawowe sposoby dokumentowania: poprzez osadzony kod HTML lub „znaczniki dokumentacyjne". *Niezależne znaczniki dokumentacyjne* są poleceniami rozpoczynającymi się od znaku `@` umieszczanymi na początku wiersza (pierwszy znak `*` w wierszu jest ignorowany). *Wewnątrzwierszowe znaczniki dokumentacyjne* mogą się pojawiać w dowolnym miejscu komentarza; podobnie jak w poprzednim przypadku rozpoczynają się one znakiem `@`, lecz dodatkowo są zapisywane w nawiasach klamrowych.

Mamy trzy rodzaje dokumentacji w komentarzach, które odpowiadają elementom poprzedzanym przez komentarz: klasie, metodzie oraz zmiennej. Tak więc komentarz opisujący klasę umieszczamy zaraz przed jej definicją, opis składowej występuje tuż przed jej definicją, a opis metody umieszcza się przed definicją tejże metody. Zerknij na prosty przykład:

```
//: object/Documentation1.java
/** Komentarz odnośnie klasy */
public class Documentation1 {
  /** Komentarz dotyczący składowej */
  public int i;
  /** Komentarz dotyczący metody */
  public void f() {}
} ///:~
```

Ważne jest to, że program *Javadoc* przerabia dokumentację jedynie dla składowych klasy typu `public` lub `protected`. Komentarze do składowych prywatnych i pakietowych są natomiast ignorowane (można jednak użyć opcji `-private` programu *Javadoc*, aby dołączyć także składowe prywatne). Jest to rozwiązanie uzasadnione, ponieważ tylko składowe publiczne i chronione z punktu widzenia użytkownika programu są dostępne spoza pliku.

W rezultacie uzyskamy plik HTML w dokładnie takim samym formacie, jak reszta dokumentacji Javy, w związku z czym użytkownicy zyskują wygodę i łatwość poruszania się po specyfikacji naszych klas. Warto zatem dopisać powyższy kod, przepuścić go przez Javadoc i obejrzeć wynikowy plik HTML.

Osadzony HTML

Kod HTML po przejściu przez Javadoc jest włączany bezpośrednio do wygenerowanej dokumentacji. Dzięki temu mamy możliwość pełnego wykorzystania składni HTML, choć podstawowym powodem jest zezwolenie na formatowanie kodu, jak pokazuje to przykład:

```
//: object/Documentation2.java
/**
* <pre>
* System.out.println(new Date());
* </pre>
*/
///:~
```

Można również stosować HTML zwyczajnie, czyli jak na każdej innej stronie internetowej w celu sformatowania normalnego tekstu będącego opisem:

```
//: object/Documentation3.java
/**
* Można <em>nawet</em> wstawić listę:
* <ol>
* <li> Element pierwszy
* <li> Element drugi
* <li> Element trzeci
* </ol>
*/
///:~
```

Wszystkie gwiazdki umieszczone na początku wierszy, podobnie jak występujące znaki spacji, są przez Javadoc odrzucane. Wiele rzeczy jest formatowanych automatycznie w celu dostosowania do standardowego wyglądu dokumentacji. Toteż nie należy przykładowo stosować znaczników <h1> czy <hr>, gdyż spowodują one niezgodność z nagłówkami, które dodaje program Javadoc.

Wszystkie wspomniane rodzaje dokumentacji umieszczanej w komentarzach — dla klas, zmiennych i metod — pozwalają na zamieszczanie wewnątrz nich kodu HTML.

Niektóre znaczniki dokumentacyjne

Poniżej przedstawiłem niektóre znaczniki, których można używać do tworzenia dokumentacji. Nim spróbujesz wykorzystać program Javadoc do wykonania jakiegoś poważnego zadania, powinieneś zapoznać się z informacjami na jego temat zamieszczonymi w dokumentacji JDK, aby w pełni zrozumieć, w jakim celu jest on stosowany.

@see

Wszystkie trzy typy dokumentacji (klasy, zmiennej i metody) mogą zawierać znacznik @see, który pozwala na odwołanie się do dokumentacji innej klasy. W miejsce znacznika @see javadoc wygeneruje kod HTML z odpowiednimi odnośnikami do innych opisów.

```
@see nazwaklasy
@see pełne-określenie-nazwyklasy
@see pełne-określenie-nazwyklasy#nazwa-metody
```

Każdy znacznik dodaje odnośnik w sekcji „See Also" wygenerowanej dokumentacji. Javadoc nie sprawdza odnośników, które podajemy, toteż warto się upewnić, czy są one poprawne.

{@link pakiet.klasa#składowa etykieta}

Podobny do znacznika `@see`, z tym, że może być stosowany wewnątrz dowolnego wiersza komentarza. Pozwala na utworzenie odnośnika z określoną etykietką w miejscu innym niż sekcja „See Also".

{@docRoot}

Generuje względną ścieżkę do głównego katalogu zawierającego dokumentację. Znacznik jest przydatny przy tworzeniu jawnych odwołań do stron należących do hierarchii katalogów zawierających dokumentację.

{@inheritDoc}

Umieszcza w bieżącym komentarzu dokumentację najbliższej klasy bazowej aktualnie dokumentowanej klasy.

@version

Znacznik ten ma postać:

```
@version informacja-o-wersji
```

gdzie `informacja-o-wersji` jest jakąś istotną informacją, którą chcemy dołączyć. Jeśli jako argument programu Javadoc wywołanego z wiersza poleceń podamy `-version`, to informacja na temat wersji zostanie zamieszczona w wygenerowanej dokumentacji.

@author

Znacznik ma postać:

```
@author informacja-o-autorze
```

gdzie `informacja-o-autorze` to przypuszczalnie nazwisko autora lub też dodatkowo adres e-mail lub inne stosowne informacje. W przypadku wywołania Javadoc z dodatkową opcją `-author` informacje te będą zamieszczane w generowanej dokumentacji.

Istnieje też możliwość zamieszczenia wielu znaczników `@author` dla kilku autorów, ale wtedy należy postępować w sposób konsekwentny. Całość informacji będzie bowiem zamieszczana łącznie w pojedynczym akapicie wygenerowanego pliku HTML.

@since

Znacznik ten pozwala wskazać wersję kodu, który rozpoczyna stosowanie danej właściwości. Można to zobaczyć w dokumentacji HTML dla Javy jako oznaczenie stosowanej wersji JDK.

@param

Znacznik `@param` jest używany przy tworzeniu dokumentacji metod i ma następującą postać:

```
@param nazwa-parameteru opis
```

`nazwa-parametru` jest dokładnym identyfikatorem parametru podanym w liście argumentów metody, a `opis` to po prostu tekst, który może zawierać wiele wierszy. Przyjmuje się, że opis kończy się wraz z kolejnym znacznikiem dokumentowania kodu. Liczba znaczników tego typu jest dowolna, lecz prawie zawsze na jeden parametr przypada jeden taki znacznik.

@return

Jest używany przy tworzeniu dokumentacji metod i ma postać:

```
@return opis
```

gdzie `opis` przedstawia znaczenie zwracanej przez metodę wartości i również może się rozciągać na wiele wierszy.

@throws

Temat wyjątków zostanie przedstawiony w rozdziale 9., jednak ogólnie mówiąc, są to obiekty, które można „wyrzucić" z metody, kiedy coś się nie powiedzie. Mimo iż tylko jeden wyjątek może się pojawić podczas wywołania metody, to konkretna metoda może oczywiście generować wiele różnych typów wyjątków, a każdy z nich powinien być opisany. Tak więc postać znacznika wyjątku jest następująca:

```
@throws pełne-określenie-nazwyklasy opis
```

gdzie `pełne-określenie-nazwyklasy` jednoznacznie określa nazwę klasy wyjątku, która jest gdzieś zdefiniowana, a `opis` (może być zamieszczony w wielu wierszach) określa powód, dla którego dany wyjątek może się pojawić.

@deprecated

Znacznik ten jest stosowany do oznaczenia niezalecanych właściwości — tych, które zostały wyparte przez ich ulepszone wersje. Sugeruje on, aby więcej nie używać danej właściwości, gdyż kiedyś w przyszłości może zostać usunięta. Metody oznaczone przez `@deprecated` sprawiają, że kompilator wypisuje je jako ostrzeżenia w przypadku, gdy są nadal używane. W Javie SE5 znacznik `@deprecated` został zastąpiony przez *adnotację* `@Deprecated` (owym adnotacjom poświęcony zostanie osobny rozdział).

Przykład dokumentowania kodu

Poniżej zamieszczony jest kolejny program w Javie, tym razem po dodaniu komentarzy dokumentujących:

```
//: object/HelloDate.java
import java.util.*;

/** Pierwszy program przykładowy z 'Thinking in Java'.
 * Wyświetla ciąg i dzisiejszą datę.
 * @author Bruce Eckel
 * @author www.MindView.net
 * @version 4.0
*/
public class HelloDate {
  /** Punkt wejścia do klasy i aplikacji.
   * @param args tablica ciągów argumentów wywołania
   * @throws exceptions nie zgłasza wyjątków
  */
  public static void main(String[] args) {
    System.out.println("Witaj, dzisiaj jest: ");
    System.out.println(new Date());
  }
} /* Output: (55% match)
Witaj, dzisiaj jest:
Fri Mar 03 16:27:29 CET 2006
*///:~
```

Pierwszy wiersz pliku to mój własny pomysł użycia znaków `//:` jako specjalnego znacznika wiersza komentarza zawierającego nazwę pliku źródłowego. W wierszu tym zawarta jest również ścieżka dostępu (w tym przypadku *object* oznacza bieżący rozdział, poświęcony obiektom w programach Javy). Ostatni wiersz także kończę komentarzem, lecz oznacza on koniec kodu źródłowego. Dzięki temu mam możliwość automatycznej aktualizacji kodu w treści książki po sprawdzeniu go przez kompilator i wykonaniu.

Znacznik `/* Output:` sygnalizuje początek oczekiwanego wyniku uruchomienia programu. W takiej postaci pozwala na automatyzację testów weryfikujących poprawność wyjścia generowanego przez program. Tutaj zapis `(55% match)` sygnalizuje systemowi testującemu, że wyjście może różnić się znacznie od oczekiwanego — system powinien spodziewać się 55-procentowej zgodności. Większość przykładów z tej książki, które w ogóle wypisują coś na wyjściu, będzie zawierać ten komentarz — dzięki niemu łatwo sprawdzisz poprawność działania programu.

Styl programowania

Nieoficjalny standard zapisu kodu w Javie, zamieszczony w dokumencie *Code Conventions for the Java Programming Language*[8], zakłada pisanie pierwszej litery w nazwie klasy jako wielkiej. Jeżeli nazwa klasy składa się z kilku słów, to pisze się je łącznie (nie stosuje się znaku podkreślenia do rozdzielenia), również rozpoczynając każde słowo wielką literą, tak jak to pokazuje przykład:

```
class WszystkieKoloryTęczy { // ...
```

[8] Dokument ten można znaleźć na stronie *http://java.sun.com/docs/codeconv/index.html*. W celu ograniczenia objętości kodów oraz możliwości ich prezentacji na seminariach nie stosowałem się do wszystkich z zaleceń wymienionych w tym dokumencie.

W każdym innym przypadku, tj. dla metod i pól (składowych klasy) oraz nazw referencji, przyjęty styl jest dokładnie taki sam *poza* tym, że pierwsza litera jest mała. Oto przykład:

```
class WszystkieKoloryTęczy {
  int liczbaCałkowitaKolorów;
  void zmieńOdcieńKoloru(int nowyOdcień) {
    // ...
  }
  // ...
}
```

Oczywiście należy pamiętać, że użytkownik będzie również musiał podawać wszystkie długie nazwy, które stworzysz, toteż zlituj się nad nim.

Kod Javy zamieszczony w bibliotekach firmy Sun również stosuje określone reguły dotyczące zamieszczania otwierających i zamykających nawiasów klamrowych, tak jak widać to w przykładach.

Podsumowanie

Celem tego rozdziału było przedstawienie informacji niezbędnych do zrozumienia sposobu, w jaki należy pisać proste programy w Javie. Zamieściłem w nim także przegląd języka oraz jego podstawowe założenia. Dotychczasowe programy miały formę: „Zrób to, następnie tamto, a potem jeszcze coś innego". Następny rozdział będzie poświęcony najważniejszym operatorom wykorzystywanym w programowaniu w Javie; potem zainteresujemy się sterowaniem wykonaniem programu.

Ćwiczenia

Normalnie ćwiczenia będą rozrzucone w treści rozdziałów, ale skoro tym razem dopiero się uczyłeś, jak pisać proste programy w nowym języku, wszystkie ćwiczenia zostały zgrupowane na końcu.

Liczba w nawiasie przy każdym ćwiczeniu reprezentuje stopień trudności ćwiczenia w skali od 1 do 10.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide* dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Ćwiczenie 1. Utwórz klasę zawierającą składowe typu `int` i `char`, które nie zostaną zainicjalizowane; wypisz wartości składowych na wyjściu programu — przekonasz się, czy Java domyślnie inicjalizuje składowe (2).

Ćwiczenie 2. Na podstawie przykładu `HelloDate.java` zamieszczonego w tym rozdziale napisz program „Witaj, świecie", który po prostu wypisuje taki napis. Wystarczy Ci tylko jedna metoda w klasie (o nazwie „main" — ta wywoływana podczas startu programu). Pamiętaj, aby uczynić ją metodą statyczną oraz dołączyć listę argumentów, nawet jeśli nie

będziesz ich używał. Skompiluj program za pomocą *javac* i uruchom, stosując *java*. Jeśli wykorzystujesz inne środowisko programowania niż JDK, to zapoznaj się z procesem kompilacji i uruchomienia w tym środowisku (1).

Ćwiczenie 3. Odnajdź fragmenty kodu dotyczące `NazwaTypu` i zrób z nich program, który da się skompilować i uruchomić (1).

Ćwiczenie 4. Zamień fragmenty kodów `TylkoDane` w program, który można skompilować i uruchomić (1).

Ćwiczenie 5. Zmodyfikuj ćwiczenie 3. tak, aby wartości zmiennych w klasie `TylkoDane` były przypisywane i wypisywane w metodzie `main()` (1).

Ćwiczenie 6. Napisz program, który zawiera i wywołuje metodę `wielkość()` zdefiniowaną w tym rozdziale (2).

Ćwiczenie 7. Zmień fragmenty kodów `StaticFun` w działający program (1).

Ćwiczenie 8. Napisz program, który udowodni, że niezależnie od liczby obiektów pewnej klasy w programie występuje tylko jeden egzemplarz wybranej składowej statycznej takiej klasy (3).

Ćwiczenie 9. Napisz program ilustrujący działanie automatycznego pakowania wartości typów podstawowych w obiekty — program ma wykorzystywać wszystkie typy podstawowe i odpowiadające im klasy (2).

Ćwiczenie 10. Napisz program, który wypisuje trzy argumenty pobrane z wiersza poleceń. Aby to zrobić, będziesz musiał indeksować tablicę typu `String` uzyskaną z wiersza poleceń (2).

Ćwiczenie 11. Zmień przykład `WszystkieKoloryTęczy` w program, który się kompiluje i działa (1).

Ćwiczenie 12. Odnajdź kod drugiego przykładu *HelloDate.java*, będący prostym przykładem dokumentowania poprzez komentarze. Wywołaj dla niego program *Javadoc* i przejrzyj wynik w przeglądarce internetowej (2).

Ćwiczenie 13. Przetwórz programem *Javadoc* pliki *Documentation1.java*, *Documentation2.java* i *Documentation3.java*. Sprawdź wynik przetwarzania w przeglądarce internetowej (1).

Ćwiczenie 14. Dodaj do dokumentacji z ćwiczenia 10. listę HTML (dowolnych elementów) (1).

Ćwiczenie 15. Do programu z ćwiczenia 2. dołącz dokumentację (w postaci komentarzy). Stwórz dokumentację w formacie HTML, stosując *Javadoc* i wyświetl w swojej przeglądarce internetowej (1).

Ćwiczenie 16. Odszukaj plik *Overloading.java* z rozdziału „Inicjalizacja i sprzątanie" i dodaj do niego komentarze dokumentacyjne. Posługując się programem *Javadoc*, wygeneruj dokumentację dla tego pliku i wyświetl ją w przeglądarce internetowej (1).

Rozdział 3.

Operatory

Na najniższym poziomie wszelkie manipulacje na danych w Javie sprowadzają się do użycia operatorów.

Ponieważ Java wywodzi się niejako z języka C++, więc większość tych operatorów będzie znana programistom C i C++. Jednak Java wprowadza też kilka poprawek i uproszczeń.

Czytelnicy znający składnię języków C i C++ mogą pokusić się o pobieżne przejrzenie tego i następnego rozdziału, skupiając się jedynie na elementach odróżniających te języki od Javy. Z kolei tych, których lektura tego rozdziału wprowadzi w zakłopotanie, odsyłam do multimedialnego seminarium *Thinking in C*, dostępnego do pobrania (nieodpłatnie) ze strony *www.MindView.net*. Seminarium zawiera wykłady, slajdy, ćwiczenia i zadania zaprojektowane specjalnie pod kątem przyswojenia podstaw niezbędnych do opanowania języka Java.

Prosta instrukcja wyjścia

W poprzednim rozdziale zapoznałeś się z podstawową instrukcją wyjścia w języku Java:

```
System.out.println("Sporo pisania");
```

Widać od razu, że wypisanie komunikatu wymaga sporo pisania (można nadwerężyć sobie ścięgna). Trudno się też tak rozwlekły kod czyta. W większości języków programowania, zarówno starszych, jak i młodszych od Javy, instrukcje wyjścia, jako często używane, zostały odpowiednio uproszczone.

Rozdział „Kontrola dostępu" wprowadza koncepcję *importu statycznego*, będącą nowością w Java SE5. Przy okazji tworzona jest tam prościutka biblioteka mająca właśnie upraszczać instrukcję wypisywania komunikatów na wyjściu programu. Nie musisz jednak znać tajników implementacji tej biblioteki, żeby zacząć z niej korzystać. Przykład z poprzedniego rozdziału przepisany z jej użyciem wyglądałby tak:

```
//: operators/HelloDate.java
import java.util.*;
import static net.mindview.util.Print.*;

public class HelloDate {
```

```
  public static void main(String[] args) {
    print("Witaj, dzisiaj jest: ");
    print(new Date());
  }
} /* Output: (55% match)
Witaj, dzisiaj jest:
Fri Mar 03 16:27:29 CET 2006
*///:~
```

Prawda, że znacznie lepiej? Warto zwrócić uwagę na pojawienie się słowa kluczowego `static` przy drugiej instrukcji `import`.

Aby korzystać z opisywanej bibliotcki, należy z witryny *www.MindView.net* (albo dowolnego z serwerów lustrzanych) pobrać kod źródłowy towarzyszący tej książce, rozpakować archiwum kodu i dodać główny katalog uzyskanej struktury katalogów do ciągu zmiennej środowiskowej `CLASSPATH` (wkrótce przedstawię pełne omówienie znaczenia tej zmiennej, tymczasem warto się do niej przyzwyczaić; problemy z jej nieodpowiednią wartością to jedne z częstszych codziennych bolączek programistów Javy).

Choć stosowanie biblioteki `net.mindview.util.Print` znakomicie upraszcza część kodu, nie nadaje się do stosowania wszędzie. Jeśli program ma wypisywać komunikaty tylko raz albo jedynie kilkukrotnie, najlepiej darować sobie dodatkowy import i po prostu wpisać instrukcję `System.out.println()`.

Ćwiczenie 1. Napisać program, który korzysta z instrukcji wyjścia w wersji skróconej i zwykłej (1).

Używanie operatorów Javy

Operator przyjmuje jeden lub więcej argumentów i zwraca nową wartość. Argumenty operatora są zapisane w innej formie niż normalne wywołania metod, ale rezultat ich działania jest ten sam. Wcześniejsze doświadczenia programistyczne powinny być wystarczające, jeśli chodzi o znajomość ogólnej idei operatorów. Dodawanie (+), odejmowanie i jednoargumentowy minus (-), mnożenie (*), dzielenie (/) oraz podstawianie (=) mają podobne działanie w każdym języku programowania.

Wszystkie operatory zwracają wartość obliczoną na podstawie swoich argumentów. Dodatkowo operator może zmienić wartość operandu. Nazywane jest to *efektem ubocznym*. Najczęściej używa się operatorów zmieniających swoje operandy po to, żeby otrzymać efekt uboczny, ale należy pamiętać, że można użyć wartości zwracanej przez taki operator tak samo, jak w przypadku operatorów nie mających efektów ubocznych.

Prawie wszystkie operatory działają tylko na typach podstawowych. Wyjątkiem są =, == oraz !=, które działają z wszystkimi obiektami (i stosowane z nimi są częstym źródłem pomyłek). Dodatkowo klasa `String` obsługuje + i +=.

Kolejność operatorów

Sposób obliczania wartości wyrażenia zawierającego kilka operatorów określają priorytety operatorów. Java ma określoną kolejność obliczeń. Najłatwiej zapamiętać, że mnożenie i dzielenie są wykonywane przed dodawaniem i odejmowaniem. Programiści często zapominają o pozostałych regułach, więc należy używać nawiasów, aby wprost określić kolejność obliczeń. W ramach przykładu przyjrzyj się instrukcjom oznaczonym jako (1) i (2):

```
//: operators/Precedence.java

public class Precedence {
  public static void main(String[] args) {
    int x = 1, y = 2, z = 3;
    int a = x + y - 2/2 + z;          // (1)
    int b = x + (y - 2)/(2 + z);      // (2)
    System.out.println("a = " + a + " b = " + b);
  }
} /* Output:
a = 5 b = 1
*///:~
```

Obie wyglądają podobnie, ale na wyjściu widać, że mają istotnie różne znaczenie, zależne właśnie od umiejscowienia nawiasów.

Zauważ, że instrukcja `System.out.println()` również angażuje operator '+'. W tym kontekście '+' oznacza „konkatenację ciągów" z ewentualną (jeśli jest konieczna) konwersją wartości na ciąg znaków. Kiedy kompilator napotyka w kodzie obiekt klasy `String`, a za nim znak '+' i wartość typu innego niż `String`, podejmuje próbę konwersji owej wartości na postać obiektu klasy `String`. Na wyjściu programu widać, że udało się tu skonwertować wartości `a` i `b` typu `int` na ciągi `String`.

Przypisanie

Przypisanie wykonywane jest przez operator =. Oznacza on: „Weź wartość prawej strony (nazywaną *p-wartością*) i skopiuj ją na lewą stronę (nazywaną *l-wartością*)". Prawa strona może być stałą, zmienną lub wyrażeniem, które zwraca wartość, ale lewa strona może być jedynie nazwą zmiennej (to znaczy musi istnieć fizyczna przestrzeń do zapamiętania wartości). Na przykład można przypisać wartość stałą do zmiennej

```
a = 4
```

ale nie można niczego przypisać do wartości stałej, więc nie może ona stać po lewej stronie (nie można napisać `4 = a;`).

Przypisania typów podstawowych są całkiem proste. Ponieważ taki typ przechowuje rzeczywistą wartość, a nie referencję do obiektu, to przypisując typ podstawowy, kopiuje się zawartość z jednego miejsca w inne. Na przykład napisanie `a = b` w przypadku

typów podstawowych skopiuje zawartość b do a. Jeśli następnie zmieni się a, nie wpłynie to na b. Jako programiści w większości sytuacji spodziewamy się właśnie takiego zachowania.

Przypisywanie obiektów wygląda jednak inaczej. Przy manipulowaniu obiektem manipulacja jest wykonywana na referencji, zatem przypisanie „jednego obiektu do drugiego" spowoduje skopiowanie referencji z jednego miejsca do drugiego. Oznacza to, że w przypadku obiektów napisanie c = d spowoduje, że zarówno c, jak i d zaczną wskazywać na obiekt, na który początkowo wskazywało tylko d. Pokazuje to poniższy przykład:

```
//: operators/Assignment.java
// Przypisania obiektów są trochę bardziej skomplikowane...
import static net.mindview.util.Print.*;

class Tank {
  int level;
}

public class Assignment {
  public static void main(String[] args) {
    Tank t1 = new Tank();
    Tank t2 = new Tank();
    t1.level = 9;
    t2.level = 47;
    print("1: t1.level: " + t1.level +
          ", t2.level: " + t2.level);
    t1 = t2;
    print("2: t1.level: " + t1.level +
          ", t2.level: " + t2.level);
    t1.level = 27;
    print("3: t1.level: " + t1.level +
          ", t2.level: " + t2.level);
  }
} /* Output:
1: t1.level: 9, t2.level: 47
2: t1.level: 47, t2.level: 47
3: t1.level: 27, t2.level: 27
*///:~
```

Klasa Tank jest prosta, a jej dwa egzemplarze (n1 oraz n2) są tworzone wewnątrz funkcji main(). Składowe level obu obiektów otrzymują różne wartości, a następnie t2 jest przypisywane do t1 oraz t1 jest zmieniane. W wielu językach programowania można się spodziewać, że t1 i t2 będą przez cały czas niezależne, ale ponieważ przypisujemy referencję, zmiana obiektu t1 spowodowała najwyraźniej zmianę obiektu t2! Stało się tak, ponieważ zarówno t1, jak i t2 zawierają tę samą referencję, która wskazuje na ten sam obiekt (oryginalna referencja w t1, wskazująca na obiekt przechowujący liczbę 9, została nadpisana przy przypisaniu i efektywnie została utracona; wskazywany przez nią obiekt zostanie usunięty w ramach automatycznego odśmiecania).

Zjawisko to jest często nazywane *tworzeniem nazwy* (ang. *aliasing*, tworzenie nowej nazwy dla istniejącego obiektu) i jest podstawą obsługi obiektów w Javie. Co zrobić, aby nie wystąpiło zjawisko aliasingu? Można zrezygnować z przypisania obiektów i zamiast tego napisać:

```
t1.level = t2.level;
```

Wtedy zamiast odrzucenia jednego z obiektów i związania „nazw" t1 i t2 z tym samym obiektem zachowane zostaną dwa oddzielne obiekty. Jednak szybko można odkryć, że bezpośrednie manipulowanie polami obiektów jest niełądne i sprzeczne z zasadami projektowania obiektowego. Temat ten nie jest trywialny — należy zawsze pamiętać, że wyniki przypisywania obiektów mogą być zaskakujące.

Ćwiczenie 2. Utworzyć klasę zawierającą składową typu `float` i użyć ją w demonstracji aliasingu obiektów (1).

Tworzenie nazw w wywołaniach metod

Aliasing wystąpi również przy przekazywaniu obiektu jako argumentu metody:

```
//: operators/PassObject.java
// Przekazywanie obiektów do metod może dać
// nieoczekiwane wyniki.
import static net.mindview.util.Print.*;

class Letter {
  char c;
}

public class PassObject {
  static void f(Letter y) {
    y.c = 'z';
  }
  public static void main(String[] args) {
    Letter x = new Letter();
    x.c = 'a';
    print("1: x.c: " + x.c);
    f(x);
    print("2: x.c: " + x.c);
  }
} /* Output:
1: x.c: a
2: x.c: z
*///:~
```

W wielu językach programowania operacje wewnątrz metody `f()` wykonywane byłyby na kopii argumentu `Letter y`. Jednak ponownie mamy do czynienia z przekazywaniem referencji, więc wiersz:

```
y.c = 'z';
```

w rzeczywistości zmieni obiekt istniejący na zewnątrz metody `f()`.

Problem aliasów (dodatkowych nazw) i jego rozwiązanie jest złożoną kwestią, która doczekała się omówienia w osobnym suplemencie (dostępnym online). Powinieneś jednak być świadomy jego istnienia oraz zwracać uwagę na związane z nim pułapki.

Ćwiczenie 3. Utwórz klasę zawierającą składową typu `float` i wykorzystaj ją w ilustracji aliasingu w wywołaniach metod (1).

Operatory matematyczne

Podstawowe operatory matematyczne Javy są takie same, jak te dostępne w większości języków programowania: dodawanie (+), odejmowanie (-), dzielenie (/), mnożenie (*) oraz modulo (%), które zwraca resztę z dzielenia całkowitego. Dzielenie całkowite obcina, a nie zaokrągla wynik.

Java używa również skróconego zapisu do równoczesnego wykonania operacji i przypisania. Oznaczane jest to przez operator, za którym znajduje się znak równości, i działa dla wszystkich operatorów języka (gdziekolwiek ma to sens). Na przykład, aby dodać 4 do zmiennej x i przypisać wynik do x, wystarczy napisać: x+=4.

Poniższy przykład pokazuje użycie operatorów matematycznych:

```
// operators/MathOps.java
// Demonstracja operatorów matematycznych.
import java.util.*;
import static net.mindview.util.Print.*;

public class MathOps {
  public static void main(String[] args) {
    // Utworzenie i inicjalizacja generatora liczb losowych:
    Random rand = new Random(47);
    int i, j, k;
    // Losowanie wartości z zakresu od 1 do 100:
    j = rand.nextInt(100) + 1;
    print("j : " + j);
    k = rand.nextInt(100) + 1;
    print("k : " + k);
    i = j + k;
    print("j + k : " + i);
    i = j - k;
    print("j - k : " + i);
    i = k / j;
    print("k / j : " + i);
    i = k * j;
    print("k * j : " + i);
    i = k % j;
    print("k % j : " + i);
    j %= k;
    print("j %= k : " + j);
    // Testy na liczbach zmiennoprzecinkowych:
    float u, v, w; // Tak samo zadziałałyby wartości typu double
    v = rand.nextFloat();
    print("v : " + v);
    w = rand.nextFloat();
    print("w : " + w);
    u = v + w;
    print("v + w : " + u);
    u = v - w;
    print("v - w : " + u);
    u = v * w;
    print("v * w : " + u);
    u = v / w;
```

```
    print("v / w : " + u);
    // Poniższe testy zadziałałyby również dla wartości typów
    // char, byte, short, int, long, i double:
    u += v;
    print("u += v : " + u);
    u -= v;
    print("u -= v : " + u);
    u *= v;
    print("u *= v : " + u);
    u /= v;
    print("u /= v : " + u);
  }
} /* Output:
j : 59
k : 56
j + k : 115
j - k : 3
k / j : 0
k * j : 3304
k % j : 56
j %= k : 3
v : 0.5309454
w : 0.0534122
v + w : 0.5843576
v - w : 0.47753322
v * w : 0.028358962
v / w : 9.940527
u += v : 10.471473
u -= v : 9.940527
u *= v : 5.2778773
u /= v : 9.940527
*///:~
```

Aby wygenerować liczby, program najpierw tworzy obiekt klasy `Random`. Gdyby jego tworzeniu nie towarzyszyły żadne argumenty, system wykonawczy użyłby w roli argumentu bieżącego czasu systemowego i nim zainicjalizował zarodek generatora liczb losowych. Jednak w przykładach prezentowanych w tej książce liczy się zgodność wyników generowanych przez programy na wyjściu (kontrolowanych zautomatyzowanymi narzędziami testującymi). Przekazując przy tworzeniu obiektu klasy `Random` wartość zarodka (wartość inicjalizującą generator losowy ustalającą jego stan i pozwalającą na odtwarzanie generowanych sekwencji liczb), wymuszamy jednakowe sekwencje liczb losowych niezależnie od momentu uruchomienia programu[1]. Aby uzyskać różne wyniki w kolejnych przebiegach programu, wystarczy zrezygnować z przekazywania argumentu przy tworzeniu obiektu klasy `Random`.

Program generuje kilka liczb różnych typów, wywołując odpowiednie metody obiektu `Random`: `nextInt()`, `nextLong()`, `nextFloat()` lub `nextDouble()`. Argument metody `nextInt()` ustawia górną granicę przedziału losowania (tak samo działa argument w wywołaniach `nextLong()` i `nextDouble()`). Granica dolna to zawsze zero; wylosowanie zera, choć mało prawdopodobne, mogłoby spowodować błąd dzielenia przez zero, więc do otrzymanej wartości dodajemy jeden.

Ćwiczenie 4. Napisz program obliczający prędkość na podstawie stałej wartości odległości i stałej wartości czasu (2).

[1] Liczba 47 cieszyła się na mojej uczelni sławą liczby „magicznej" — i tak już zostało.

Jednoargumentowe operatory minus i plus

Jednoargumentowy minus (-) i plus (+) wyglądają tak samo, jak ich dwuargumentowe odpowiedniki. Kompilator odgaduje, którego z nich należy użyć, na podstawie składni wyrażenia. Na przykład wyrażenie:

```
x = -a;
```

ma oczywiste znaczenie. Kompilator jest również w stanie odgadnąć sens:

```
x = a * -b;
```

ale osoba czytająca kod może być zdezorientowana, więc lepiej napisać:

```
x = a * (-b);
```

Jednoargumentowy minus daje wartość przeciwną. Jednoargumentowy plus jest wprowadzony w celu symetrii z minusem, ale nie ma żadnego efektu.

Operatory zwiększania i zmniejszania

Java, tak jak C, zawiera wiele ułatwień. Sprawiają one, że łatwiej jest pisać kod Javy, ale z jego czytelnością bywa różnie.

Dwoma przyjemnymi skrótami są operatory zwiększania i zmniejszania (często nazywane operatorami autoinkrementacji i autodekrementacji). Operator zmniejszania to `--` i oznacza on: „Zmniejsz o jednostkę". Operator zwiększania to `++` i oznacza: „Zwiększ o jednostkę". Jeśli na przykład a jest liczbą całkowitą typu `int`, to wyrażenie `++a` jest równoważne z `(a=a+1)`. Operatory zwiększania i zmniejszania zwracają jako wynik wartość zmiennej.

Istnieją dwie wersje operatora każdego typu, często nazywane wersją przedrostkową (pre-) i wersją przyrostkową (post-). Preinkrementacja oznacza, że operator `++` pojawia się przed zmienną lub wyrażeniem, a postinkrementacja oznacza, że operator `++` pojawia się po zmiennej lub wyrażeniu. Podobnie predekrementacja oznacza, że operator `--` pojawia się przed zmienną lub wyrażeniem, a postdekrementacja oznacza, że operator `--` pojawia się po zmiennej lub wyrażeniu. Przy preinkrementacji i predekrementacji (tj. `++a` lub `--a`) najpierw wykonywana jest operacja, a następnie zwracany jej wynik. Przy postinkrementacji i postdekrementacji (tj. `a++` lub `a--`) najpierw zwracany jest wynik, a następnie wykonywana jest operacja. Na przykład:

```
// operators/AutoInc.java
// Demonstracja działania operatorów ++ i --.
import static net.mindview.util.Print.*;

public class AutoInc {
  public static void main(String[] args) {
    int i = 1;
    print("i : " + i);
    print("++i : " + ++i); // Preinkrementacja
    print("i++ : " + i++); // Postinkrementacja
    print("i : " + i);
```

```
    print("--i : " + --i); // Predekrementacja
    print("i-- : " + i--); // Postdekrementacja
    print("i : " + i);
  }
} /* Output:
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
*///:~
```

Widać, że dla postaci przedrostkowej wartość zwracana jest po wykonaniu operacji, natomiast dla postaci przyrostkowej wartość jest zwracana przed wykonaniem operacji. Są to jedyne operatory (oprócz tych związanych z przypisaniami), które mają efekty uboczne (to znaczy zmieniają operand, a nie jedynie wykorzystują jego wartość).

Operator zwiększania w pewien sposób wyjaśnia nazwę C++, sugerującą „jeden krok ponad C". We wczesnym wystąpieniu dotyczącym Javy Bill Joy (jeden z jej twórców) powiedział, że „Java = C++—" (C plus plus minus minus), wskazując, że Java to C++, z którego usunięto niepotrzebne trudne elementy i tym samym stała się ona językiem o wiele prostszym niż C++. Zapoznając się z tą książką, przekonasz się, że wiele elementów jest prostszych, ale Java nie jest przez to uboższa od C++.

Operatory relacji

Operatory relacji zwracają wartości logiczne, tj. typu `boolean`. Służą do oceny związku pomiędzy wartościami operandów. Wyrażenie relacyjne zwraca `true`, jeśli relacja jest prawdziwa, oraz `false`, jeśli relacja jest fałszywa. Operatorami relacyjnymi są: mniejszy (<), większy (>), mniejszy lub równy (<=), większy lub równy (>=), równość (==) oraz nierówność (!=). Równość i nierówność działają ze wszystkimi typami podstawowymi, natomiast pozostałe operatory porównania nie będą działały z typem logicznym `boolean`. Ponieważ `boolean` może mieć wartość `true` albo `false`, nie sposób dla takiego typu zdefiniować sensownie relacji „mniejsze niż" czy „większe niż".

Sprawdzanie równości obiektów

Operatory logiczne == i != działają również ze wszystkimi obiektami, ale ich działanie często zaskakuje początkujących programistów Javy. Spójrzmy na przykład:

```
//: operators/Equivalence.java

public class Equivalence {
  public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1 == n2);
    System.out.println(n1 != n2);
  }
```

```
} /* Output:
false
true
*///:~
```

Wyrażenie `System.out.println(n1 == n2)` wypisze wynik wyrażenia znajdującego się wewnątrz nawiasów. Można się spodziewać, że wynikiem wykonania tego przykładu będą wartości `true` i `false`, ponieważ obydwa obiekty są takie same. Ale podczas gdy *zawartość* obiektów jest taka sama, to referencje do nich nie są te same, a operatory `==` i `!=` porównują referencje do obiektów. Więc w rzeczywistości wynikiem jest `false` i `true`. Naturalnie początkowo może to być zaskakujące.

Jeśli konieczne jest sprawdzenie równości zawartości obiektów, należy użyć specjalnej metody `equals()` zdefiniowanej dla wszystkich obiektów (ale nie typów podstawowych, dla których dobrze działają `==` i `!=`). Oto przykład jej zastosowania:

```
//: operators/EqualsMethod.java

public class EqualsMethod {
  public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1.equals(n2));
  }
} /* Output:
true
*///:~
```

Tak, jak się można było spodziewać, wynikiem będzie wartość `true`. Nie jest to jednak takie proste. W przypadku klasy stworzonej samodzielnie, tak jak poniżej:

```
//: operators/EqualsMethod2.java
// Domyślna wersja equals() nie porównuje zawartości.

class Value {
  int i;
}

public class EqualsMethod2 {
  public static void main(String[] args) {
    Value v1 = new Value();
    Value v2 = new Value();
    v1.i = v2.i = 100;
    System.out.println(v1.equals(v2));
  }
} /* Output:
false
*///:~
```

wracamy do początku — wynikiem jest `false`. Jest tak, ponieważ domyślnie `equals()` porównuje referencje. Więc jeśli metoda `equals()` nie zostanie *przesłonięta* w nowej klasie, nie będzie można uzyskać żądanego zachowania. Niestety, przesłanianie poznasz dopiero w rozdziale „Wielokrotne wykorzystanie klas", a odpowiedni sposób definiowania metody `equals()` zostanie omówiony dopiero w rozdziale „Kontenery", ale uprzednia znajomość działania `equals()` może zaoszczędzić trochę kłopotów.

Większość klas biblioteki standardowej Javy implementuje `equals()` tak, że porównywana jest ich zawartość, a nie referencje.

Ćwiczenie 5. Utwórz klasę o nazwie `Dog` (pies) zawierającą dwie składowe typu `String`: `name` (imię psa) i `says` („daj głos"). W funkcji `main()` utwórz dwa obiekty tej klasy, o nazwach `spot` (który na komendę „daj głos" reaguje szczekiem „Hau") i `scruffy` (który warczy „Wrrr"). Wypisz na wyjściu imiona psów i głos, jaki psy wydają (2).

Ćwiczenie 6. Kontynuując ćwiczenie 5., utwórz nową referencję obiektu klasy `Dog` i przypisz ją do obiektu `spot`. Porównaj ze sobą (operatorem == i metodą `equals()`) wszystkie referencje `Dog` występujące w programie (3).

Operatory logiczne

Operatory logiczne: koniunkcja (&&), alternatywa (||) oraz negacja (!) zwracają wartość logiczną „prawda" lub „fałsz", określoną przez logiczną relację ich argumentów. W poniższym przykładzie zostały użyte operatory relacyjne i logiczne:

```
//: operators/Bool.java
// Operatory relacyjne i logiczne.
import java.util.*;
import static net.mindview.util.Print.*;

public class Bool {
  public static void main(String[] args) {
    Random rand = new Random(47);
    int i = rand.nextInt(100);
    int j = rand.nextInt(100);
    print("i = " + i);
    print("j = " + j);
    print("i > j to " + (i > j));
    print("i < j to " + (i < j));
    print("i >= j to " + (i >= j));
    print("i <= j to " + (i <= j));
    print("i == j to " + (i == j));
    print("i != j to " + (i != j));
    // W Javie nie można traktować wartości boolean jak wartości int:
//!  print("i && j to " + (i && j));
//!  print("i || j to " + (i || j));
//!  print("!i to " + !i);
    print("(i < 10) && (j < 10) to "
       + ((i < 10) && (j < 10)) );
    print("(i < 10) || (j < 10) to "
       + ((i < 10) || (j < 10)) );
  }
} /* Output:
i = 58
j = 55
i > j to true
i < j to false
i >= j to true
i <= j to false
i == j to false
```

```
i != j to true
(i < 10) && (j < 10) to false
(i < 10) || (j < 10) to false
*///:~
```

Koniunkcję, alternatywę i negację można stosować tylko dla argumentów typu logicznego. Nie można, tak jak w C i C++, używać innych typów danych w ten sam sposób jak typów logicznych. Nieudane próby zastosowania tego sposobu można zaobserwować w wierszach wykomentowanych przez znacznik //! (pozwalający zautomatyzowanemu systemowi na usuwanie takich komentarzy w celu wykonania i przetestowania oznaczonych nimi wierszy). Jednakże kolejne wyrażenia zwracają już wartości logiczne, bo używają relacji porównania. Można zatem użyć operatorów logicznych na ich wynikach.

Jeśli wartość logiczna zostanie użyta w wyrażeniu w miejscu, gdzie spodziewany jest typ `String`, to jest ona automatycznie zamieniana na odpowiednią postać tekstową.

W powyższym programie w definicjach typu `int` można wstawić dowolny inny typ podstawowy oprócz logicznego, tj. `boolean`. Należy jednak uważać, ponieważ porównywanie liczb zmiennopozycyjnych jest bardzo precyzyjne. Liczba różniąca się nawet minimalnie od innej liczby nadal „nie jest jej równa". Liczba tylko odrobinę różniąca się od zera nadal nie jest zerem.

Ćwiczenie 7. Napisz program, który symuluje zabawę w rzucanie monetą (3).

Skracanie obliczenia wyrażenia logicznego

Zajmując się operatorami logicznymi, można natknąć się na zjawisko nazywane „skracaniem". Działa on w ten sposób, że wyrażenie będzie przetwarzane, *dopóki* jego prawdziwość lub fałszywość nie zostanie jasno określona. W rezultacie niektóre części wyrażenia logicznego mogą nie zostać przetworzone. Oto przykład demonstrujący mechanizm skracania:

```
// : operators/ShortCircuit.java
// Demonstruje działanie skróconej ewaluacji wyrażeń
// z operatorami logicznymi.
import static net.mindview.util.Print.*;

public class ShortCircuit {
  static boolean test1(int val) {
    print("test1(" + val + ")");
    print("wynik: " + (val < 1));
    return val < 1;
  }
  static boolean test2(int val) {
    print("test2(" + val + ")");
    print("wynik: " + (val < 2));
    return val < 2;
  }
  static boolean test3(int val) {
    print("test3(" + val + ")");
    print("wynik: " + (val < 3));
    return val < 3;
  }
```

```
  public static void main(String[] args) {
    boolean b = test1(0) && test2(2) && test3(2);
    print("wyrażenie ma wartość " + b);
  }
} /* Output:
test1(0)
wynik: true
test2(2)
wynik: false
wyrażenie ma wartość false
*///:~
```

Każdy test wykonuje porównanie argumentów i zwraca „prawdę" lub „fałsz". Wypisuje również informacje, aby pokazać, że porównanie zostało wykonane. Testy są używane w wyrażeniu:

```
test1(0) && test2(2) && test3(2)
```

Oczywiście mogłoby się wydawać, że wykonane zostaną wszystkie testy, ale wyniki programu wskazują na coś innego. Pierwszy wynik zwrócił „prawdę", zatem przetwarzanie wyrażenia jest kontynuowane. Jednakże drugi test zwraca „fałsz". Ponieważ oznacza to, że całe wyrażenie musi być fałszywe, po co kontynuować przetwarzanie reszty wyrażenia? Może to być kosztowne. I właśnie to jest powodem skracania — wydajność może wzrosnąć, jeśli nie ma konieczności przetwarzania wszystkich części wyrażenia logicznego.

Literały

Zwykle po wpisaniu literału do programu kompilator wie dokładnie, jaki typ danych ma mu nadać. Jednakże czasami typ może nie być jednoznacznie określony. Jeśli tak się stanie, należy pokierować kompilatorem przez podanie dodatkowych informacji w postaci znaków połączonych z wartością literału. Poniższy kod pokazuje te znaki:

```
//: operators/Literals.java
import static net.mindview.util.Print.*;

public class Literals {
  public static void main(String[] args) {
    int i1 = 0x2f; // Literał szesnastkowy (z małymi literami)
    print("i1: " + Integer.toBinaryString(i1));
    int i2 = 0X2F; // Literał szesnastkowy (z wielkimi literami)
    print("i2: " + Integer.toBinaryString(i2));
    int i3 = 0177; // Literał ósemkowy (z zerem na początku)
    print("i3: " + Integer.toBinaryString(i3));
    char c = 0xffff; // Szesnastkowy literał maksymalnej wartości char
    print("c: " + Integer.toBinaryString(c));
    byte b = 0x7f; // Szesnastkowy literał maksymalnej wartości byte
    print("b: " + Integer.toBinaryString(b));
    short s = 0x7fff; // Szesnastkowy literał maksymalnej wartości short
    print("s: " + Integer.toBinaryString(s));
    long n1 = 200L; // Przyrostek wartości long
    long n2 = 200l; // Przyrostek wartości long (mylący)
    long n3 = 200;
    float f1 = 1;
```

```
    float f2 = 1F; // Przyrostek wartości float
    float f3 = 1f; // Przyrostek wartości float
    double d1 = 1d; // Przyrostek wartości double
    double d2 = 1D; // Przyrostek wartości double
    // (Można też dla typu long stosować literały szesnastkowe i ósemkowe)
  }
} /* Output:
i1: 101111
i2: 101111
i3: 1111111
c: 111111111111111
b: 1111111
s: 111111111111111
*///:~
```

Znak uzupełniający wartość literału określa jego typ. Małe i wielkie litery `L` oznaczają typ całkowity `long`, przy czym stosowanie małego `l` może być mylące, bo takie `l` wygląda w kodzie jak jedynka. Małe i wielkie `F` oznacza typ zmiennoprzecinkowy `float`. Małe i wielkie `D` oznacza zaś typ `double`.

Zapis szesnastkowy (kod liczbowy o podstawie 16), który działa dla wszystkich typów całkowitych, rozpoczyna się od `0x` lub `0X`, za którymi następują cyfry 0 – 9 i małe lub wielkie litery a – f. Gdy spróbuje się zainicjować zmienną przez wartość większą, niż może ona pomieścić (niezależnie od numerycznego zapisu tej wartości), kompilator zwróci komunikat o błędzie. W powyższym kodzie użyto maksymalnych możliwych szesnastkowych wartości dla `char`, `byte` i `short`. Po ich przekroczeniu kompilator automatycznie zamieni typ wartości na `int` i zwróci informację, że do przypisania konieczne jest rzutowanie z obcięciem (o rzutowaniu powiemy za chwilę). Wiadomo wtedy, że limit został przekroczony.

Liczba w zapisie ósemkowym (o podstawie 8) rozpoczyna się zerem i zawiera cyfry 0 – 7.

W Javie, C i C++ nie ma literałów reprezentujących liczby binarne. Ale przy pracy z zapisem szesnastkowym i ósemkowym niekiedy pożądane jest prezentowanie wyników w postaci binarnej. Ciąg takiej reprezentacji można łatwo uzyskać za pomocą statycznej metody `toBinaryString()` klas `Integer` i `Long`. Zauważ, że przy przekazywaniu do `Integer.toBinaryString()` wartości typu mniejszego od `int` wymuszamy automatyczną konwersję tej wartości na typ `int`.

Ćwiczenie 8. Sprawdź, czy zapis ósemkowy i szesnastkowy można stosować dla wartości typu `long`. Do wypisania wyników użyj metody `Long.toBinaryString()` (2).

Zapis wykładniczy

Wykładniki zapisywane są w postaci, którą zawsze uważałem za szczególnie mylącą:

```
//: operators/Exponents.java
// "e" oznacza "10 do potęgi".

public class Exponents {
  public static void main(String[] args) {
    // Małe i wielkie 'e' mają to samo znaczenie:
    float expFloat = 1.39e-43f;
    expFloat = 1.39E-43f;
```

```
    System.out.println(expFloat);
    double expDouble = 47e47d; // 'd' jest opcjonalne
    double expDouble2 = 47e47; // Automatyczna konwersja na typ double
    System.out.println(expDouble);
  }
} /* Output:
1.39E-43
4.7E48
*///:~
```

W naukach ścisłych i inżynierii „e" oznacza podstawę logarytmu naturalnego, w przybliżeniu równą 2,718 (bardziej dokładna wartość typu `double` jest dostępna w Javie jako `Math.E`). Używa się jej w wyrażeniach wykładniczych, takich jak $1{,}39 * e^{-43}$, co oznacza $1{,}39 * 2{,}718^{-43}$. Jednak kiedy wynaleziono FORTRAN, zadecydowano, że e będzie oznaczać „dziesięć do potęgi". Jest to co najmniej dziwne, ponieważ FORTRAN był projektowany dla naukowców i inżynierów, więc mogłoby się wydawać, że jego twórcy będą uważać, by nie wprowadzić takiej niejednoznaczności[2]. W każdym razie zwyczaj ten przeszedł do C i C++, a teraz również do Javy. Jeśli więc przyzwyczajony jesteś do myślenia o e jako podstawie logarytmu naturalnego, to będziesz musiał przeprowadzać myślowe tłumaczenie za każdym razem, gdy zobaczy w Javie wyrażenie typu `1.39e-43f` — oznacza ono $1{,}39 * 10^{-43}$.

Nie ma konieczności dodawania znaku na końcu literałów, jeśli kompilator może sam odgadnąć odpowiedni typ. Przy zapisie:

```
long n3 = 200;
```

nie ma nieporozumień, więc `L` po 200 byłoby nadmiarowe. Mimo to przy:

```
float f4 = 1e-43f; // 10 do potęgi
```

kompilator normalnie określa stałe zmiennopozycyjne jako `double`, więc brak kończącego `f` spowoduje wystąpienie błędu i trzeba użyć rzutowania, by zamienić `double` na `float`.

Ćwiczenie 9. Wypisać największą i najmniejszą wartość typu `float` i `double` w zapisie wykładniczym (1).

[2] John Kirkham pisze: „Zacząłem programować w 1962 r., używając FORTRANA II na maszynie IBM 1620. W tym czasie oraz przez lata 60. aż do 70. FORTRAN był w całości językiem pisanym wielkimi literami. Zaczęło się to najprawdopodobniej od tego, że wczesnymi urządzeniami wejściowymi były stare jednostki teletekstu, które używały pięciobitowego kodu *Baudot* uniemożliwiającego pisanie małymi literami. Również „E" w zapisie wykładniczym zawsze było pisane wielką literą, więc nigdy nie było mylone z podstawą logarytmu naturalnego „e", które jest zawsze pisane małą literą. Po prostu „E" oznaczało podstawę potęgi o wartości równej podstawie używanego systemu liczbowego — najczęściej 10. W tym czasie system ósemkowy był również często używany przez programistów. Mimo że sam nigdy tego nie widziałem, to gdybym jednak zobaczył liczbę ósemkową w zapisie wykładniczym, przyjąłbym, że podstawą potęgi jest 8. Pamiętam, że pierwszy raz zobaczyłem małe „e" w zapisie wykładniczym pod koniec lat 70. i również wydało mi się to mylące. Problem powstał, kiedy małe znaki zakradły się do FORTRANU, a nie od jego początku. Właściwie to mieliśmy funkcje, których używaliśmy, kiedy potrzebna nam była podstawa logarytmu naturalnego, ale one wszystkie były pisane wielkimi literami".

Operatory bitowe

Operatory bitowe pozwalają manipulować pojedynczymi bitami argumentów całkowitych. Wynik operatorów bitowych jest obliczany przez wykonanie operacji algebry logicznej na odpowiadających sobie bitach obydwu argumentów.

Operatory bitowe wywodzą się z niskopoziomowego rozwiązania znanego z C. Często dokonywano w tym języku bezpośrednich manipulacji sprzętowych, gdy trzeba było ustawiać bity rejestrów urządzeń. Java była pierwotnie projektowana dla zastosowań w przystawkach telewizyjnych, więc to niskopoziomowe zorientowanie nie jest całkowicie pozbawione sensu. Jednakże najprawdopodobniej nie będziesz musiał używać takich operatorów zbyt często.

Bitowy operator koniunkcji (&) ustawia wyjściowy bit na jeden, jeśli obydwa bity wejściowe są jedynkami. W przeciwnym razie ustawia zero. Bitowy operator alternatywy (|) ustawia bit wyjściowy na jeden, jeśli którykolwiek z bitów wejścia jest jedynką, a na zero tylko wtedy, kiedy obydwa bity wejściowe są zerem. Bitowy operator alternatywy wykluczającej XOR (^) ustawia bit wyjściowy na jedynkę, jeśli jeden z bitów wejściowych jest jedynką, ale nie wtedy, gdy obydwa mają tę samą wartość. Bitowy operator negacji (~), nazywany również operatorem *uzupełnienia do jedynki*, jest operatorem unarnym — przyjmuje tylko jeden argument (wszystkie pozostałe operatory bitowe są dwuargumentowe). Bitowa negacja zwraca przeciwieństwo bitu wejściowego — jedynkę, jeśli bit wejściowy jest zerem, zero — jeśli bit wejściowy jest jedynką.

Operatory bitowe i logiczne używają tych samych znaków, więc aby zapamiętać ich znaczenie, można zastosować następującą mnemotechnikę: ponieważ bity są „małe", to w operatorach bitowych jest tylko jeden znak.

Operatory bitowe można łączyć ze znakiem =, aby połączyć operację z przypisaniem. Zatem wszystkie &=, |= oraz ^= są legalne (ponieważ ~ jest operatorem jednoargumentowym, nie może być łączony ze znakiem =).

Typ logiczny jest traktowany jako jednobitowy, więc trochę się różni. Można wykonać bitowe AND, OR i XOR, ale nie można wykonać bitowego NOT (przypuszczalnie dlatego, by uniemożliwić pomylenie go z logicznym NOT). Dla argumentów typu logicznego operatory bitowe działają tak samo jak logiczne, z wyjątkiem tego, że nie ulegają skracaniu. Operacje bitowe na argumentach logicznych obejmują XOR; operator bitowy, który nie ma odpowiednika na liście operatorów „logicznych". Jak opisałem to poniżej, zabronione jest używanie zmiennych logicznych w wyrażeniach przesuwania.

Ćwiczenie 10. Napisz program z dwoma wartościami stałymi. Pierwsza z nich ma zawierać na przemian binarne jedynki i zera, z zerem na najmniej znaczącej pozycji. Druga również ma zawierać przeplatankę jedynek i zer, ale z jedynką na najmniej znaczącej pozycji (podpowiedź: takie wartości najłatwiej wyrażać w systemie szesnastkowym). Wartości te poddaj wszystkim kombinacjom wywołań operatorów binarnych; wyniki wypisz za pośrednictwem metody `Integer.toBinaryString()` (3).

Operatory przesunięć

Operatory przesunięć również manipulują bitami. Mogą być używane wyłącznie z typami podstawowymi, całkowitymi. Operator przesunięcia w lewo (<<) zwraca operand znajdujący się na lewo od niego, przesunięty o liczbę bitów podaną po operatorze (wstawiając zera na najmłodszych bitach). Operator przesunięcia w prawo ze znakiem (>>) zwraca operand znajdujący się na lewo od niego, przesunięty w prawo o liczbę bitów podaną po operatorze. Przesunięcie w prawo ze znakiem (oznaczane >>) używa *rozszerzania znaku*, tzn. jeśli wartość jest dodatnia, w miejsce najstarszych bitów wstawiane są zera; jeśli wartość jest ujemna, w miejsce najstarszych bitów wstawiane są jedynki. Java wprowadza również przesunięcie w prawo bez znaku (>>>), które używa *rozszerzania zerem*, tzn. niezależnie od znaku w miejsce najstarszych bitów wstawiane są zera. Ten operator nie istnieje w C i C++.

Przy przesuwaniu argumenty typu `char`, `byte` lub `short` zostaną najpierw rozszerzone do `int` i wynik również będzie typu `int`. Użytych zostanie tylko pięć najmłodszych bitów operandu z prawej strony. Zapobiega to przesunięciu o więcej bitów, niż jest ich w typie `int`. Operując na argumentach typu `long`, otrzymuje się wynik typu `long`. Użyte będzie tylko sześć najmłodszych bitów prawej strony, więc nie można przesunąć o więcej bitów, niż jest ich w typie `long`.

Przesunięcie można łączyć ze znakiem równości (<<= lub >>= lub >>>=). Wtedy lewa strona jest zastępowana przez swą wartość przesuniętą o wartość prawej strony operatora. Jednakże problemem jest połączenie przypisania i przesunięcia w prawo bez znaku. Jeśli użyje się go z argumentem typu `byte` lub `short`, wyniki nie będą poprawne. Zamiast tego są one rozszerzane do typu `int` i przesuwane w prawo, a następnie obcinane, ponieważ są przypisywane z powrotem do swoich zmiennych. Istnieją przypadki, w których wynikiem będzie -1. Pokazuje to następujący przykład:

```
//: operators/URShift.java
// Test przesunięć w prawo bez znaku.
import static net.mindview.util.Print.*;

public class URShift {
  public static void main(String[] args) {
    int i = -1;
    print(Integer.toBinaryString(i));
    i >>>= 10;
    print(Integer.toBinaryString(i));
    long l = -1;
    print(Long.toBinaryString(l));
    l >>>= 10;
    print(Long.toBinaryString(l));
    short s = -1;
    print(Integer.toBinaryString(s));
    s >>>= 10;
    print(Integer.toBinaryString(s));
    byte b = -1;
    print(Integer.toBinaryString(b));
    b >>>= 10;
    print(Integer.toBinaryString(b));
```

```
    b = -1,
    print(Integer.toBinaryString(b));
    print(Integer.toBinaryString(b>>>10));
  }
} /* Output:
11111111111111111111111111111111
11111111111111111111
11111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111
*///:~
```

W ostatnim przesunięciu wartość wynikowa nie jest przypisywana z powrotem do b, tylko wypisywana bezpośrednio, więc zachowanie będzie prawidłowe.

Oto przykład demonstrujący użycie wszystkich operatorów związanych z bitami:

```
//: operators/BitManipulation.java
// Stosowanie operatorów bitowych.
import java.util.*;
import static net.mindview.util.Print.*;

public class BitManipulation {
  public static void main(String[] args) {
    Random rand = new Random(47);
    int i = rand.nextInt();
    int j = rand.nextInt();
    printBinaryInt("-1", -1);
    printBinaryInt("+1", +1);
    int maxpos = 2147483647;
    printBinaryInt("maxpos", maxpos);
    int maxneg = -2147483648;
    printBinaryInt("maxneg", maxneg);
    printBinaryInt("i", i);
    printBinaryInt("~i", ~i);
    printBinaryInt("-i", -i);
    printBinaryInt("j", j);
    printBinaryInt("i & j", i & j);
    printBinaryInt("i | j", i | j);
    printBinaryInt("i ^ j", i ^ j);
    printBinaryInt("i << 5", i << 5);
    printBinaryInt("i >> 5", i >> 5);
    printBinaryInt("(~i) >> 5", (~i) >> 5);
    printBinaryInt("i >>> 5", i >>> 5);
    printBinaryInt("(~i) >>> 5", (~i) >>> 5);

    long l = rand.nextLong();
    long m = rand.nextLong();
    printBinaryLong("-1L", -1L);
    printBinaryLong("+1L", +1L);
    long ll = 9223372036854775807L;
    printBinaryLong("maxpos", ll);
    long lln = -9223372036854775808L;
```

```
    printBinaryLong("maxneg", lln);
    printBinaryLong("l", l);
    printBinaryLong("~l", ~l);
    printBinaryLong("-l", -l);
    printBinaryLong("m", m);
    printBinaryLong("l & m", l & m);
    printBinaryLong("l | m", l | m);
    printBinaryLong("l ^ m", l ^ m);
    printBinaryLong("l << 5", l << 5);
    printBinaryLong("l >> 5", l >> 5);
    printBinaryLong("(~l) >> 5", (~l) >> 5);
    printBinaryLong("l >>> 5", l >>> 5);
    printBinaryLong("(~l) >>> 5", (~l) >>> 5);
  }
  static void printBinaryInt(String s, int i) {
    print(s + ", int: " + i + ", binarnie:\n   " +
      Integer.toBinaryString(i));
  }
  static void printBinaryLong(String s, long l) {
    print(s + ", long: " + l + ", binarnie:\n    " +
      Long.toBinaryString(l));
  }
} /* Output:
-1, int: -1, binary:
  11111111111111111111111111111111
+1, int: 1, binary:
  1
maxpos, int: 2147483647, binarnie:
  1111111111111111111111111111111
maxneg, int: -2147483648, binarnie:
  10000000000000000000000000000000
i, int: -1172028779, binarnie:
  10111010001001000100001010010101
~i, int: 1172028778, binarnie:
  01000101110110111011110101101010
-i, int: 1172028779, binarnie:
  01000101110110111011110101101011
j, int: 1717241110, binarnie:
  01100110010110110000010100010110
i & j, int: 570425364, binarnie:
  00010000000000000000000000010100
i | j, int: -25213033, binarnie:
  11111110011111111010001111010111
i ^ j, int: -595638397, binarnie:
  11011100011111111010001111000011
i << 5, int: 1149784736, binarnie:
  01000100010000010100101010100000
i >> 5, int: -36625900, binarnie:
  11111101110100010010001000010100
(~i) >> 5, int: 36625899, binarnie:
  00010111011011011011101011
i >>> 5, int: 97591828, binarnie:
  00101110100010010001000010100
(~i) >>> 5, int: 36625899, binarnie:
  00010111011011011011101011
...
*///:~
```

Ostatnie dwie metody, `printBinaryInt()` oraz `printBinaryLong()`, przyjmują odpowiednio argumenty typu `int` oraz `long` i drukują je w postaci binarnej razem z łańcuchem opisu. Oprócz demonstrowania efektów wszystkich operatorów bitowych dla `int` i `long` przykład ten pokazuje, jak wyglądają wartości: minimum, maksimum, +1 i –1 dla typów `int` i `long`. Widać również, że najstarszy bit reprezentuje znak liczby: zero oznacza liczbę dodatnią, a jedynka — ujemną. Fragment oczekiwanego wyjścia dla typu `int` widać na przykładzie.

Taka binarna reprezentacja liczb jest nazywana *kodem uzupełnieniowym do dwóch ze znakiem*.

Ćwiczenie 11. Zacznij od liczby, która w zapisie binarnym ma jedynkę na najbardziej znaczącej pozycji bitowej (wskazówka: najlepiej skorzystać z zapisu szesnastkowego); przy użyciu operatora przesunięcia w prawo ze znakiem przesuwaj tę jedynkę na kolejne pozycje bitowe, za każdym razem wypisując wynik na wyjściu za pomocą metody `Integer.toBinaryString()` (3).

Ćwiczenie 12. Zacznij od liczby, która w zapisie binarnym ma jedynki na wszystkich pozycjach bitowych. Przesuń ją w lewo, a następnie przy użyciu operatora przesunięcia w prawo bez znaku przesuwaj wszystkie bity liczby na kolejne pozycje, za każdym razem wypisując wynik na wyjściu za pomocą metody `Integer.toBinaryString()` (3).

Ćwiczenie 13. Napisz metodę, która wypisze wartość typu `char` (znak) w postaci binarnej. Przetestuj ją przy użyciu kilku różnych znaków (1).

Operator trójargumentowy if-else

Ten operator, zwany też *operatorem warunkowym*, jest inny niż pozostałe, ponieważ przyjmuje trzy operandy. Jest to prawdziwy operator, gdyż zwraca wartość w przeciwieństwie do zwykłej instrukcji `if-else`, która zostanie przedstawiona w następnym rozdziale. Jest to wyrażenie w postaci:

```
wyrażenie-logiczne ? wartość0 : wartość1
```

Jeśli `wyrażenie-logiczne` przyjmie wartość „prawda", to wyliczane jest wyrażenie `wartość0` i operator zwraca jego wynik. Jeśli `wyrażenie-logiczne` przyjmie wartość „fałsz", to wyliczane jest wyrażenie `wartość1` i operator zwraca jego wynik.

Można oczywiście używać normalnej instrukcji `if-else` (opisanej w następnym rozdziale), ale operator trójargumentowy jest bardziej zwięzły. Mimo że dumą C (bo stąd pochodzi ten operator) jest jego zwięzłość — a operator trójargumentowy został być może wprowadzony częściowo z powodu wydajności — należy go używać ostrożnie, gdyż łatwo jest stworzyć kod nieczytelny.

Operator trójargumentowy różni się od instrukcji `if-else` tym, że zwraca wartość. Oto przykład porównujący obie konstrukcje:

```
//: operators/TernaryIfElse.java
import static net.mindview.util.Print.*;
```

```
public class TernaryIfElse {
  static int ternary(int i) {
    return i < 10 ? i * 100 : i * 10;
  }
  static int standardIfElse(int i) {
    if(i < 10)
      return i * 100;
    else
      return i * 10;
  }
  public static void main(String[] args) {
    print(ternary(9));
    print(ternary(10));
    print(standardIfElse(9));
    print(standardIfElse(10));
  }
} /* Output:
900
100
900
100
*///:~
```

Widać, że kod metody `ternary()` jest bardziej zwięzły niż konkurencyjny kod pozbawiony operatora trójargumentowego, widoczny w metodzie `standardIfElse()`. Ale metoda `standardIfElse()` jest łatwiejsza do ogarnięcia i wcale nie wymaga wiele więcej pisania. Zawsze, kiedy wybiera się operator trójargumentowy, należy rozważyć, czy są do tego wystarczające powody — najczęściej stosuje się go jedynie tam, gdzie trzeba ustawić zmienną na jedną z dwóch wartości.

Operatory + i += dla klasy String

W Javie istnieje jedno specjalne użycie operatora: operatory + i += można używać do sklejania (konkatenacji) ciągów znaków. Takie użycie obu operatorów wydaje się naturalne, mimo że nie jest zgodne z tradycyjnym.

Ta możliwość wydawała się dobrym pomysłem w C++, więc w C++ dodano możliwość *przeciążania operatorów*, pozwalającą programiście nadać inne znaczenie niemal wszystkim operatorom. Niestety, przeciążanie operatorów w połączeniu z innymi ograniczeniami C++ okazało się dość skomplikowaną cechą przy projektowaniu własnych klas. Przeciążanie operatorów byłoby o wiele łatwiejsze do zaimplementowania w Javie niż w C++ (czego dowodzi język C#, w którym *można* w prosty sposób przeciążać operatory). Mimo to cecha ta nadal uważana jest za zbyt skomplikowaną, więc programiści Javy nie mogą implementować przeciążonych operatorów tak, jak mogą to robić programiści C++ i C#.

Omawiane operatory, kiedy są używane dla obiektów klasy `String`, przejawiają ciekawe zachowanie. Otóż, jeśli wyrażenie zaczyna się od ciągu znaków `String`, to wszystkie kolejne operandy również muszą być ciągami (pamiętaj, że kompilator automatycznie zamieni literał łańcuchowy zapisany w cudzysłowie na typ `String`):

```
//: operators/StringOperators.java
import static net.mindview.util.Print.*;

public class StringOperators {
  public static void main(String[] args) {
    int x = 0, y = 1, z = 2;
    String s = "x, y, z ";
    print(s + x + y + z);
    print(x + " " + s); // Konwersja x na obiekt klasy String
    s += "(zsumowane) = "; // Operator konkatenacji
    print(s + (x + y + z));
    print("" + x); // Skrót od Integer.toString()
  }
} /* Output:
x, y, z 012
0 x, y, z
x, y, z (zsumowane) = 3
0
*///:~
```

Wyjście generowane przez pierwszą instrukcję to „012" zamiast najzwyklejszego '3', a przecież przy sumowaniu liczb całkowitych oczekujemy właśnie sumowania wartości, a nie konkatenacji ciągów reprezentujących te liczby. Tutaj kompilator Javy, zamiast dodać wartości x, y oraz z, zamieni je na odpowiadające im reprezentacje tekstowe i połączy w jeden łańcuch. W drugiej instrukcji wyjścia pierwszy z przekazanych argumentów zostanie skonwertowany na typ `String`; jak widać, konwersja jest niezależna od postaci pierwszej wartości. Wreszcie mamy tu użycie operatora `+=` do dołączenia ciągu do zmiennej `s`, a potem kontrolę kolejności obliczania argumentów wywołania za pośrednictwem nawiasów; w tym ostatnim przypadku zgrupowane w nawiasie wartości typu `int` zostaną zsumowane, a dopiero suma zostanie skonwertowana na ciąg i wyświetlona.

Zwróć uwagę na ostatnią instrukcję w metodzie `main()`: niekiedy widać w kodzie takie konstrukcje z pustym ciągiem na przedzie i operatorem `+`; to prosty sposób przeprowadzenia konwersji wartości na ciąg znaków bez jawnego wywoływania odpowiedniej do tego metody (tu byłaby nią metoda `Integer.toString()`).

Najczęstsze pułapki przy używaniu operatorów

Jedną z pułapek związanych z użyciem operatorów jest próba pominięcia nawiasów, gdy istnieją najmniejsze chociaż wątpliwości co do tego, jak wyrażenie będzie obliczane. W Javie problem ten pozostaje aktualny.

Niezmiernie częstym błędem w C i C++ było:

```
while(x = y) {
    // ...
}
```

Programista próbował sprawdzić równość (==), a nie wykonać przypisanie. W C i C++ wynikiem przypisania jest wartość przypisywanego wyrażenia, co w połączeniu z możliwością (a w przypadku C koniecznością) używania typów liczbowych jako typów logicznych, oznacza zawsze „prawda", jeśli y nie jest zerem. Taka pętla będzie więc nieskończona. W Javie wynik tego wyrażenia nie jest typu logicznego. Kompilator spodziewa się typu logicznego i nie zamieni `int` na `boolean`, co spowoduje błąd w trakcie kompilacji — jeszcze przed próbą uruchomienia programu. Zatem ta pułapka nigdy nie pojawi się w Javie (jednak gdy x i y są typu `boolean`, nie spowoduje to błędu kompilacji, bo wtedy x=y jest legalnym wyrażeniem, chociaż w powyższym przypadku najprawdopodobniej błędnym).

Podobnym problemem w C i C++ jest używanie bitowego AND i OR zamiast ich logicznych odpowiedników. Bitowe AND i OR używają po jednym znaku (& lub |), podczas gdy logiczne AND i OR używają dwóch (&& oraz ||). Tak samo jak z = i == łatwo jest zapisać tylko jeden znak zamiast dwóch. W Javie kompilator zapobiega temu, ponieważ nie pozwoli nonszalancko używać innego typu tam, gdzie on nie pasuje.

Operatory rzutowania

Słowo *rzutować* używane jest tutaj w sensie „rzutowania do postaci". Java w razie potrzeby automatycznie zamieni jeden typ danych na inny. Na przykład przy przypisaniu wartości całkowitej do zmiennej zmiennopozycyjnej kompilator automatycznie zamieni `int` na `float`. Rzutowanie umożliwia bezpośrednie wskazanie takiej konwersji lub wymuszenie konwersji w miejscu, w którym normalnie by nie nastąpiła.

Aby wykonać rzutowanie, należy wstawić żądany typ danych w nawiasach po lewej stronie dowolnej wartości. Oto przykład:

```
//: operators/Casting.java

public class Casting {
  public static void main(String[] args) {
    int i = 200;
    long lng = (long)i;
    lng = i; // "Poszerzanie", więc bez faktycznej potrzeby rzutowania
    long lng2 = (long)200;
    lng2 = 200;
    // "Zwężanie":
    i = (int)lng2; // Konieczne rzutowanie
  }
} ///:~
```

Jak widać, rzutowanie można wykonać zarówno na wartości liczbowej, jak i na zmiennej. Jednakże w obydwu przedstawionych wyżej przypadkach rzutowanie jest zbyteczne, ponieważ — jeśli jest ono konieczne — kompilator automatycznie promuje typ `int` do `long`. Mimo to można używać nadmiarowego rzutowania, aby coś podkreślić, albo sprawić, by kod był bardziej czytelny. W innych sytuacjach wykonanie rzutowania może być konieczne do skompilowania programu.

W C i C++ rzutowanie może czasem przyprawiać o ból głowy. W Javie rzutowanie jest bezpieczne z tym wyjątkiem, że wykonując tak zwaną *konwersję z obcięciem* (lub inaczej *konwersję zawężającą*, tzn. przechodząc z typu danych mogącego przechować więcej informacji na taki, który nie może tyle przechować), ryzykuje się utratę informacji. Tutaj kompilator wymusza bezpośrednie rzutowanie tak, jakby mówił: „To może być niebezpieczne — jeśli mam to zrobić, musisz mi bezpośrednio nakazać rzutowanie". Przy *konwersji rozszerzającej* bezpośrednie rzutowanie nie jest konieczne, ponieważ nowy typ danych może przechować więcej informacji niż stary, informacja nie zostanie więc nigdy stracona.

Java pozwala na rzutowanie każdego typu podstawowego na każdy inny typ podstawowy, oprócz typu logicznego, który w ogóle nie może być rzutowany. Typy klas również nie mogą być rzutowane. Konieczne są specjalne metody, by zamienić jedną klasę na inną (wyjątkiem jest `String` oraz — jak pokazane jest w dalszych rozdziałach tej książki — możliwe jest rzutowanie w ramach *rodziny* typów; `Dąb` może zostać zrzutowany na `Drzewo` i odwrotnie, ale nie na obcy typ, taki jak `Kamień`).

Obcinanie a zaokrąglanie

Przy wymuszaniu konwersji zawężających trzeba zwracać uwagę na kwestie obcinania i zaokrąglania. Jak będzie wyglądać przykładowe rzutowanie wartości zmiennoprzecinkowej na typ całkowity? Jak Java obsłuży konwersję wartości 29.7 na typ `int` — czy wynikiem konwersji będzie 29, czy może 30? Odpowiedź na te pytania znajdziesz w następnym przykładzie:

```
//: operators/CastingNumbers.java
// Co się stanie w wyniku rzutowania wartości
// zmiennoprzecinkowej (float albo double) na typ całkowity?
import static net.mindview.util.Print.*;

public class CastingNumbers {
  public static void main(String[] args) {
    double above = 0.7, below = 0.4;
    float fabove = 0.7f, fbelow = 0.4f;
    print("(int)above: " + (int)above);
    print("(int)below: " + (int)below);
    print("(int)fabove: " + (int)fabove);
    print("(int)fbelow: " + (int)fbelow);
  }
} /* Output:
(int)above: 0
(int)below: 0
(int)fabove: 0
(int)fbelow: 0
*///:~
```

Jak widać, rzutowanie wartości typu `float` bądź `double` na typ całkowity zawsze prowadzi do obcięcia części ułamkowej. Jeśli wartość wynikowa ma zostać zaokrąglona, należy skorzystać z metody `round()` z klasy `java.lang.Math`:

```
//: operators/RoundingNumbers.java
// Zaokrąglanie wartości zmiennoprzecinkowych.
import static net.mindview.util.Print.*;
```

```
public class RoundingNumbers {
  public static void main(String[] args) {
    double above = 0.7, below = 0.4;
    float fabove = 0.7f, fbelow = 0.4f;
    print("Math.round(above): " + Math.round(above));
    print("Math.round(below): " + Math.round(below));
    print("Math.round(fabove): " + Math.round(fabove));
    print("Math.round(fbelow): " + Math.round(fbelow));
  }
} /* Output:
Math.round(above): 1
Math.round(below): 0
Math.round(fabove): 1
Math.round(fbelow): 0
*///:~
```

Ponieważ metoda `round()` wchodzi w skład pakietu `java.lang`, nie trzeba jej klasy jawnie importować do programu.

Promocja typu

Przed wykonywaniem dowolnej operacji matematycznej lub bitowej na podstawowych typach danych mniejszych od `int` (tj. `char`, `byte` lub `short`) argumenty zostaną promowane do typu `int` i wynik będzie również typu `int`. Zatem należy użyć rzutowania, aby z powrotem przypisać je do mniejszego typu (ale przy przypisaniu do mniejszego typu można stracić część informacji). Ogólnie mówiąc, rozmiar wyniku wyrażenia określa największy typ danych występujący w wyrażeniu (przy mnożeniu typu `float` przez `double` wynik będzie typu `double`; po dodaniu `int` i `long` wynikiem będzie `long`).

W Javie nie ma „sizeof"

W C i C++ operator `sizeof()` miał specjalne zadanie: określał liczbę jednostek pamięci przydzielonych na element danych. Był on konieczny, by zapewnić przenośność programów. Typy danych na różnych maszynach mogą mieć różne rozmiary, więc programista musi mieć możliwość sprawdzenia rozmiaru typów. Na przykład, jeden komputer może przechowywać liczby całkowite w 32 bitach, podczas gdy inny w 16, czyli program na pierwszej maszynie może w liczbie całkowitej przechowywać większe wartości. Można sobie wyobrazić, jakie problemy wywoływały próby przenoszenia programów pisanych w C i C++.

Java nie potrzebuje do tego operatora `sizeof()`, ponieważ wszystkie typy danych są takie same na wszystkich maszynach. Nie ma potrzeby zajmowania się przenośnością na tym poziomie, ponieważ została ona określona w projekcie języka.

Kompendium operatorów

Poniższy przykład pokazuje, które typy danych mogą być używane z każdym z operatorów. W zasadzie jest to ten sam przykład powtórzony wielokrotnie, za każdym razem przy użyciu innego z podstawowych typów danych. Plik zostanie skompilowany bez błędów, ponieważ wiersze, które powodowałyby błędy, zostały wykomentowane przez //!.

```
// : operators/AllOps.java
// Testy wszystkich operatorów na wszystkich podstawowych
// typach danych, ilustrujące konstrukcje akceptowane przez
// kompilator języka Java.

public class AllOps {
  // Akceptuje wyniki testu logicznego:
  void f(boolean b) {}
  void boolTest(boolean x, boolean y) {
    // Operatory arytmetyczne:
    //! x = x * y;
    //! x = x / y;
    //! x = x % y;
    //! x = x + y;
    //! x = x - y;
    //! x++;
    //! x--;
    //! x = +y;
    //! x = -y;
    // Operatory relacyjne i logiczne:
    //! f(x > y);
    //! f(x >= y);
    //! f(x < y);
    //! f(x <= y);
    f(x == y);
    f(x != y);
    f(!y);
    x = x && y;
    x = x || y;
    // Operatory bitowe:
    //! x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Przypisania złożone:
    //! x += y;
    //! x -= y;
    //! x *= y;
    //! x /= y;
    //! x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    x &= y;
    x ^= y;
```

```
  x |= y;
  // Rzutowanie:
  //! char c = (char)x;
  //! byte b = (byte)x;
  //! short s = (short)x;
  //! int i = (int)x;
  //! long l = (long)x;
  //! float f = (float)x;
  //! double d = (double)x;
}
void charTest(char x, char y) {
  // Operatory arytmetyczne:
  x = (char)(x * y);
  x = (char)(x / y);
  x = (char)(x % y);
  x = (char)(x + y);
  x = (char)(x - y);
  x++;
  x--;
  x = (char)+y;
  x = (char)-y;
  // Operatory relacyjne i logiczne:
  f(x > y);
  f(x >= y);
  f(x < y);
  f(x <= y);
  f(x == y);
  f(x != y);
  //! f(!x);
  //! f(x && y);
  //! f(x || y);
  // Operatory bitowe:
  x= (char)~y;
  x = (char)(x & y);
  x  = (char)(x | y);
  x = (char)(x ^ y);
  x = (char)(x << 1);
  x = (char)(x >> 1);
  x = (char)(x >>> 1);
  // Przypisania złożone:
  x += y;
  x -= y;
  x *= y;
  x /= y;
  x %= y;
  x <<= 1;
  x >>= 1;
  x >>>= 1;
  x &= y;
  x ^= y;
  x |= y;
  // Rzutowanie:
  //! boolean bl = (boolean)x;
  byte b = (byte)x;
  short s = (short)x;
  int i = (int)x;
  long l = (long)x;
```

```
  float f = (float)x;
  double d = (double)x;
}
void byteTest(byte x, byte y) {
  // Operatory arytmetyczne:
  x = (byte)(x* y);
  x = (byte)(x / y);
  x = (byte)(x % y);
  x = (byte)(x + y);
  x = (byte)(x - y);
  x++;
  x--;
  x = (byte)+ y;
  x = (byte)- y;
  // Operatory relacji i logiczne:
  f(x > y);
  f(x >= y);
  f(x < y);
  f(x <= y);
  f(x == y);
  f(x != y);
  //! f(!x);
  //! f(x && y);
  //! f(x || y);
  // Operatory bitowe:
  x = (byte)~y;
  x = (byte)(x & y);
  x = (byte)(x | y);
  x = (byte)(x ^ y);
  x = (byte)(x << 1);
  x = (byte)(x >> 1);
  x = (byte)(x >>> 1);
  // Przypisania złożone:
  x += y;
  x -= y;
  x *= y;
  x /= y;
  x %= y;
  x <<= 1;
  x >>= 1;
  x >>>= 1;
  x &= y;
  x ^= y;
  x |= y;
  // Rzutowanie:
  //! boolean bl = (boolean)x;
  char c = (char)x;
  short s = (short)x;
  int i = (int)x;
  long l = (long)x;
  float f = (float)x;
  double d = (double)x;
}
void shortTest(short x, short y) {
  // Operatory arytmetyczne:
  x = (short)(x * y);
  x = (short)(x / y);
  x = (short)(x % y);
```

```
    x = (short)(x + y);
    x = (short)(x - y);
    x++;
    x--;
    x = (short)+y;
    x = (short)-y;
    // Operatory relacji i logiczne:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operatory bitowe:
    x = (short)~y;
    x = (short)(x & y);
    x = (short)(x | y);
    x = (short)(x ^ y);
    x = (short)(x << 1);
    x = (short)(x >> 1);
    x = (short)(x >>> 1);
    // Przypisania złożone:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Rzutowanie:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void intTest(int x, int y) {
    // Operatory arytmetyczne:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Operatory relacyjne i logiczne:
    f(x > y);
```

```
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operatory bitowe:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Przypisania złożone:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Rzutowanie:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void longTest(long x, long y) {
    // Operatory arytmetyczne:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Operatory relacyjne i logiczne:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
```

```
  // Operatory bitowe:
  x = ~y;
  x = x & y;
  x = x | y;
  x = x ^ y;
  x = x << 1;
  x = x >> 1;
  x = x >>> 1;
  // Przypisania złożone:
  x += y;
  x -= y;
  x *= y;
  x /= y;
  x %= y;
  x <<= 1;
  x >>= 1;
  x >>>= 1;
  x &= y;
  x ^= y;
  x |= y;
  // Rzutowanie:
  //! boolean bl = (boolean)x;
  char c = (char)x;
  byte b = (byte)x;
  short s = (short)x;
  int i = (int)x;
  float f = (float)x;
  double d = (double)x;
}
void floatTest(float x, float y) {
  // Operatory arytmetyczne:
  x = x * y;
  x = x / y;
  x = x % y;
  x = x + y;
  x = x - y;
  x++;
  x--;
  x = +y;
  x = -y;
  // Operatory relacyjne i logiczne:
  f(x > y);
  f(x >= y);
  f(x < y);
  f(x <= y);
  f(x == y);
  f(x != y);
  //! f(!x);
  //! f(x && y);
  //! f(x || y);
  // Operatory bitowe:
  //! x = ~y;
  //! x = x & y;
  //! x = x | y;
  //! x = x ^ y;
  //! x = x << 1;
  //! x = x >> 1;
  //! x = x >>> 1;
```

```
    // Przypisania złożone:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Rzutowanie:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    double d = (double)x;
  }
  void doubleTest(double x, double y) {
    // Operatory arytmetyczne:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Operatory relacyjne i logiczne:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operatory bitowe:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Przypisania złożone:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
```

```
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Rzutowanie:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
  }
} ///:~
```

Jak widać, typ logiczny jest dość ograniczony. Zmiennym tego typu można nadać wartość `true` lub `false` oraz sprawdzić ich prawdziwość lub fałszywość, ale nie można ich dodawać ani wykonywać na nich żadnych innych operacji oprócz operacji logicznych.

W przypadku typów `char`, `byte` i `short` widać efekty promocji przy operatorach arytmetycznych. Każda operacja arytmetyczna na każdym z tych typów daje w wyniku typ `int`, który przed przypisaniem do oryginalnego typu musi być z powrotem rzutowany (jest to konwersja z obcięciem, która może spowodować utratę informacji). Tylko przy typie `int` nie ma potrzeby rzutowania, ponieważ wszystko jest już typu `int`. Nie wszystkie działania są tu jednak bezpieczne. Jeśli pomnoży się przez siebie dwie dostatecznie duże liczby typu `int`, może wystąpić przepełnienie wyniku. Pokazuje to poniższy przykład:

```
//: operators/Overflow.java
// Niespodzianka! Java dopuszcza przepełnienie.

public class Overflow {
  public static void main(String[] args) {
    int big = Integer.MAX_VALUE;
    System.out.println("big = " + big);
    int bigger = big * 4;
    System.out.println("bigger = " + bigger);
  }
} /* Output:
big = 2147483647
bigger = -4
*///:~
```

Kompilator nie zwraca żadnego komunikatu o błędzie ani ostrzeżenia i nie ma żadnych wyjątków w trakcie wykonania. Java jest dobra, ale nie *aż* tak.

Przypisania złożone *nie wymagają* rzutowania dla typów `char`, `byte` lub `short`, mimo że dokonują promocji, które mają taki sam skutek, jak bezpośrednie operacje arytmetyczne.

Widać również, że z wyjątkiem typu logicznego każdy typ podstawowy może być rzutowany na każdy inny typ podstawowy. Należy więc pamiętać o skutkach konwersji z obcięciem przy rzutowaniu na mniejszy typ, gdyż w przeciwnym razie można w trakcie rzutowania nieświadomie stracić informacje.

Ćwiczenie 14. Napisz metodę, która przyjmie dwa argumenty typu `String` i wykorzysta wszystkie operatory logiczne do porównania tych dwóch obiektów; metoda ma wypisać wyniki poszczególnych porównań. Przy stwierdzaniu równości (==) i różności (!=) należy użyć również metody `equals()`. W metodzie `main()` należy wywołać napisaną metodę dla paru różnych obiektów klasy `String` (3).

Podsumowanie

Kto miał już styczność z językami opartymi na składni języka C, uzna operatory języka Java za na tyle znajome, że nie wymagają dłuższej nauki. Tym, dla których rozdział okazał się zbyt trudny, odsyłam do multimedialnego kursu *Thinking in C*, publikowanego w witrynie *www.MindView.net*.

Rozwiązania wybranych ćwiczeń można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 4.

Sterowanie przebiegiem wykonania

Program — tak jak żyjąca istota — musi manipulować otaczającym go światem i dokonywać wyborów w trakcie działania.

Java używa wszystkich instrukcji warunkowych znanych z C, więc jeśli programowałeś w C lub C++, to wszystko będzie znajome. Większość proceduralnych języków programowania ma jakieś instrukcje warunkowe. Wśród słów kluczowych Javy znajdują się instrukcje: `if-else`, `while`, `do-while`, `for`, `return`, `break` oraz instrukcja wyboru `switch`. Jednak Java nie obsługuje niesławnej instrukcji `goto` (która mimo to pozostaje najszybszą metodą rozwiązywania szczególnego typu problemów). Nadal można wykonywać skoki w stylu `goto`, jednak jest to o wiele bardziej sztuczne niż typowe `goto`.

Prawda i fałsz

Wszystkie instrukcje warunkowe określają kolejną instrukcję na podstawie prawdziwości lub fałszywości wyrażenia warunkowego. Wyrażeniem warunkowym jest na przykład `A==B`. Operator warunkowy == został użyty, by określić, czy wartość `A` jest równa wartości `B`. Wyrażenie zwraca `true` lub `false`. Do stworzenia instrukcji warunkowej można użyć dowolnego z przedstawionych wcześniej operatorów relacyjnych. Zauważ, że Java nie pozwala na użycie liczby zamiast wartości logicznej, mimo że jest to dozwolone w C i C++ (gdzie prawda jest niezerowa, a fałsz jest zerem). Chcąc użyć wartości innego typu niż logiczny w teście logicznym, takim jak `if(a)`, trzeba najpierw zamienić ją na wartość logiczną, używając wyrażenia warunkowego, takiego jak `if(a != 0)`.

if-else

Instrukcja warunkowa `if-else` jest podstawową metodą sterowania wykonaniem programu. Część `else` można pominąć, więc `if` ma dwie postacie:

```
if (wyrazenie logiczne)
  instrukcja
```

lub

```
if (wyrażenie-logiczne)
  instrukcja1
else
  instrukcja2
```

Warunek reprezentowany wyrażeniem logicznym musi zwracać wartość logiczną. Instrukcja oznacza zarówno prostą instrukcję zakończoną średnikiem, jak i instrukcję złożoną, czyli blok instrukcji prostych otoczonych nawiasami klamrowymi. Jeśli gdziekolwiek dalej wystąpi słowo „instrukcja", to oznacza, że można użyć instrukcji prostej lub złożonej.

Jako przykład `if-else` poniżej przedstawiona jest metoda `test()`, która sprawdza, czy liczba jest większa, mniejsza czy równa liczbie odgadywanej:

```
//: control/IfElse.java
import static net.mindview.util.Print.*;

public class IfElse {
  static int result = 0;
  static void test(int testval, int target) {
    if(testval > target)
      result = +1;
    else if(testval < target)
      result = -1;
    else
      result = 0; // Równe
  }
  public static void main(String[] args) {
    test(10, 5);
    print(result);
    test(5, 10);
    print(result);
    test(5, 5);
    print(result);
  }
} /* Output:
1
-1
0
*///:~
```

W środku metody `test()` widać też konstrukcję `else if` — nie ma tu mowy o osobnym słowie kluczowym — to po prostu klauzula `else` z instrukcją `if-else`.

Choć Java (tak jak C i C++) nie narzuca programiście żadnego stylu formatowania kodu, wygodnie jest stosować wcięcia instrukcji sterujących, aby czytający mógł łatwo określić ich początek i koniec.

Iteracja

Sterowanie wykonaniem pętli odbywa się za pośrednictwem instrukcji `while`, `do-while` i `for`; czasem są one nazywane *instrukcjami iteracji*. W każdej z pętli następuje powtarzanie `instrukcji` dopóty, dopóki `wyrażenie-logiczne` kontrolujące pętlę jest prawdziwe. Pętla `while` ma postać:

```
while (wyrażenie-logiczne)
  instrukcja
```

`Wyrażenie-logiczne` jest obliczane przed rozpoczęciem pętli, a następnie ponownie przy każdej kolejnej iteracji `instrukcji`.

W poniższym przykładzie liczby losowe są generowane do momentu, aż zajdzie określony warunek.

```
//: control/WhileTest.java
// Demonstracja pętli while.

public class WhileTest {
  static boolean condition() {
    boolean result = Math.random() < 0.99;
    System.out.print(result + ", ");
    return result;
  }
  public static void main(String[] args) {
    while(condition())
      System.out.println("Wewnątrz 'while'");
    System.out.println("Poza 'while'");
  }
} /* (Uruchom, aby zobaczyć wynik) *///:~
```

Metoda `condition()` korzysta ze statycznej metody `random()` pochodzącej z biblioteki `Math`, która generuje liczbę typu `double` pomiędzy 0 i 1 (włączając w to 0, ale bez 1). Wartość `result` to wynik zastosowania operatora relacji <, który zwraca wartość `boolean`. Wypisanie wartości typu `boolean` na wyjściu programu powoduje automatyczną konwersję tej wartości na ciąg „true" albo „false". Wyrażenie logiczne pętli `while` oznacza: „Wykonuj pętlę, dopóki `condition()` zwraca wartość `true`".

do-while

Instrukcja `do-while` ma postać:

```
do
  instrukcja
while (wyrażenie-logiczne);
```

Jedyną różnicą między `while` a `do-while` jest to, że w pętli `do-while` instrukcja jest wykonywana przynajmniej raz, nawet jeśli wyrażenie jest fałszywe od samego początku. Jeśli w pętli `while` warunek jest od początku fałszywy, instrukcja nigdy nie zostanie wykonana. W praktyce częściej spotyka się pętlę `while`.

for

To chyba najpopularniejsza instrukcja iteracji. Przed pierwszą iteracją pętli for wykonywana jest inicjalizacja. Na początku każdej iteracji sprawdzany jest warunek, a na końcu instrukcje przejścia do następnego kroku. Instrukcja for ma postać:

```
for (inicjalizacja; wyrażenie-logiczne; krok)
  instrukcja
```

Każde z wyrażeń inicjalizacja, wyrażenie-logiczne lub krok może być puste. Wyrażenie jest sprawdzane przed każdą iteracją i — jak tylko przyjmie wartość false — wykonanie będzie kontynuowane od pierwszego wiersza występującego po instrukcji for. Na koniec każdej pętli wykonywany jest krok.

Pętle for stosuje się zwykle w zadaniach „wyliczających":

```
// . control/ListCharacters.java
// Demonstruje zastosowanie pętli for w generowaniu
// wykazu kompletu małych liter zestawu ASCII.

public class ListCharacters {
  public static void main(String[] args) {
    for(char c = 0; c < 128; c++)
      if(Character.isLowerCase(c))
        System.out.println("wartość: " + (int)c +
          " znak: " + c);
  }
} /* Output:
wartość: 97 znak: a
wartość: 98 znak: b
wartość: 99 znak: c
wartość: 100 znak: d
wartość: 101 znak: e
wartość: 102 znak: f
wartość: 103 znak: g
wartość: 104 znak: h
wartość: 105 znak: i
wartość: 106 znak: j
...
*///:~
```

Jak widać, zmienna c jest zdefiniowana w tym samym miejscu, w którym jest używana — wewnątrz wyrażenia sterującego pętli for, a nie na przykład na początku metody main(). Zasięg c ogranicza się do wyrażenia kontrolowanego przez for.

Powyższy program wykorzystuje także klasę „opakowującą" java.lang.Character, która nie tylko umieszcza zmienną podstawowego typu char w obiekcie, lecz dostarcza także innych, dodatkowych możliwości. W tym przypadku statyczna metoda isLowerChar() pozwala określić, czy badany znak jest małą literą.

Tradycyjne języki programowania, takie jak C, wymagają, by wszystkie zmienne były definiowane na początku bloku, tak by kompilator, tworząc blok, mógł przydzielić dla nich miejsce. W Javie i C++ deklaracje mogą być porozrzucane po całym bloku i definiowane w miejscu, w którym są potrzebne. Pozwala to na bardziej naturalny styl kodowania i sprawia, że łatwiej taki kod zrozumieć.

Ćwiczenie 1. Napisz program, który wypisze na wyjście wartości od 1 do 100 (1).

Ćwiczenie 2. Napisz program, który wygeneruje dwadzieścia pięć wartości losowych typu `int`. Dla każdej wartości zastosuj instrukcję warunkową `if-else`, klasyfikując wartość jako większą, mniejszą albo równą innej, również losowo wygenerowanej wartości (2).

Ćwiczenie 3. Zmień program z ćwiczenia 2. tak, żeby kod był ujęty w „nieskończonej" pętli `while`. Program taki będzie działał w pętli dopóty, dopóki nie przerwiesz go naciśnięciem odpowiedniej kombinacji klawiszy (zwykle *Ctrl+C*) (1).

Ćwiczenie 4. Napisz program, który użyje dwóch zagnieżdżonych pętli `for` i operatora modulo (`%`) do wykrywania i wypisywania kolejnych liczb pierwszych (liczb całkowitych, które dzielą się bez reszty jedynie przez 1 i przez siebie same) (3).

Ćwiczenie 5. Powtórz ćwiczenie 10. z poprzedniego rozdziału, wykorzystując do wypisania na wyjściu jedynek i zer operator trójargumentowy i operatory bitowe (w odpowiedniej pętli); program ma więc samodzielnie realizować zadanie, które w pierwotnym ćwiczeniu wykonywała metoda `Integer.toBinaryString()` (4).

Operator przecinka

Przecinek jako *operator* (nie jako *separator*, używany do oddzielenia definicji i argumentów funkcji) ma w Javie tylko jedno zastosowanie — w wyrażeniu kontrolnym pętli `for`. Zarówno w części inicjalizacyjnej, jak i w kroku można podać kilka instrukcji oddzielonych przecinkami, które zostaną wykonane sekwencyjnie.

Za pomocą operatora przecinka można definiować w instrukcji `for` wiele zmiennych, byle wszystkie były tego samego typu:

```
//: control/CommaOperator.java

public class CommaOperator {
  public static void main(String[] args) {
    for(int i = 1, j = i + 10; i < 5; i++, j = i * 2) {
      System.out.println("i = " + i + " j = " + j);
    }
  }
} /* Output:
i = 1 j = 11
i = 2 j = 4
i = 3 j = 6
i = 4 j = 8
*///:~
```

Definicja typu `int` w instrukcji `for` odnosi się zarówno do `i`, jak i do `j`. Część inicjalizacyjna pętli może zawierać dowolną liczbę definicji, byle wszystkie miały *ten sam typ*. Możliwość definiowania zmiennych w wyrażeniu sterującym dotyczy jedynie pętli `for`. Nie można używać tej metody z żadną inną instrukcją wyboru lub iteracji.

Widać, że zarówno w części inicjalizującej, jak i kroku instrukcje wykonywane są w kolejności sekwencyjnej.

Składnia foreach

Java SE5 wprowadziła nową odmianę składni pętli for, przeznaczoną do przeglądania elementów kontenerów i tablic (którym poświęcone są zresztą osobne rozdziały: „Tablice" i „Kontenery z bliska"). Odmiana ta jest często określana mianem składni foreach i nie wymaga tworzenia specjalnej zmiennej int reprezentującej licznik iteracji — foreach zajmuje się tym automatycznie.

Za przykład posłuży tablica wartości typu float; załóżmy, że chcemy przejrzeć zawartość tablicy, odwołując się do kolejnych elementów:

```
//: control/ForEachFloat.java
import java.util.*;

public class ForEachFloat {
  public static void main(String[] args) {
    Random rand = new Random(47);
    float f[] = new float[10];
    for(int i = 0; i < 10; i++)
      f[i] = rand.nextFloat();
    for(float x : f)
      System.out.println(x);
  }
} /* Output:
0.72711575
0.39982635
0.5309454
0.0534122
0.16020656
0.57799757
0.18847865
0.4170137
0.51660204
0.73734957
*///:~
```

Tablica jest wypełniana za pomocą klasycznej pętli for, bo przy wstawianiu elementów trzeba jawnie podać indeks w tablicy (potrzebujemy do tego licznika pętli). Składnia foreach pokazana jest w wierszu:

```
for (float x : f)
```

Powyższa instrukcja definiuje zmienną x typu float i przypisuje jej kolejne wartości pobierane z naszej tablicy.

Składnia foreach nadaje się do stosowania z dowolnymi metodami, które w wyniku zwracają tablice. Przykładem może być klasa String, której metoda toCharArray() zwraca tablicę znaków (wartości char); można więc w prosty sposób przejrzeć kolejne znaki ciągu:

```
//: control/ForEachString.java

public class ForEachString {
  public static void main(String[] args) {
    for(char c : "Jaskółka afrykańska".toCharArray() )
      System.out.print(c + " ");
```

```
  }
} /* Output:
Jaskółka afrykańska
*///:~
```

W rozdziale „Kolekcje obiektów" dowiesz się, że składnię foreach można stosować z dowolnym obiektem, który jest obiektem `Iterable`.

Wiele instrukcji `for` polega na przechodzeniu sekwencji kolejnych wartości całkowitych, jak tu:

```
for (int i = 0; i < 100; i++)
```

Składnia foreach tu nie zadziała, chyba że uprzednio utworzymy stuelementową tablicę wartości typu `int`. Aby rzecz uprościć, utworzyłem w pakiecie `net.mindview.util.Range` metodę o nawie `range()`, która automatycznie generuje odpowiednią tablicę. Metodę `range()` tworzyłem z myślą o imporcie statycznym:

```
//: control/ForEachInt.java
import static net.mindview.util.Range.*;
import static net.mindview.util.Print.*;

public class ForEachInt {
  public static void main(String[] args) {
    for(int i : range(10)) // 0..9
      printnb(i + " ");
    print();
    for(int i : range(5, 10)) // 5..9
      printnb(i + " ");
    print();
    for(int i : range(5, 20, 3)) // 5..20 z krokiem 3
      printnb(i + " ");
    print();
  }
} /* Output:
0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
5 8 11 14 17
*///:~
```

Metoda `range()` została przeciążona, co oznacza, że występuje pod tą samą nazwą dla różnych list argumentów (o przeciążaniu powiemy sobie wkrótce). Pierwsza przeciążona wersja metody `range()` zaczyna od zera i generuje wartości aż do zadanej argumentem granicy zakresu (ale bez wartości granicznej). Druga wersja zaczyna od wartości pierwszego argumentu i generuje wartości aż do wartości drugiego argumentu (ale bez niej); trzecia wersja uwzględnia rozmiar kroku w sekwencji wartości. Metoda `range()` jest bardzo prostą wersją tak zwanego *generatora*, o których później.

Zauważ, że choć `range()` zwiększa zakres zastosowania składni foreach, co z kolei niewątpliwie przyczynia się do zwiększenia czytelności kodu, to stosowanie takiego połączenia jest odrobinę mniej efektywne od zwykłych pętli `for`; tam, gdzie ważna jest wydajność, najlepiej zmierzyć stratę za pomocą programu profilującego, mierzącego wydajność kodu.

Poza metodą `print()` pojawiła się tu metoda `printnb()`. Ta ostatnia nie uzupełnia wypisywanego komunikatu znakiem nowego wiersza, pozwala więc na wypisywanie wierszy „na raty".

Składnia foreach niewątpliwie pozwala na zaoszczędzenie na ilości wpisywanego kodu, a więc i na czasie. Ale, co ważniejsze, jest znacznie czytelniejsza od instrukcji for, bo jawnie wyraża zamiar programisty (przejrzenia każdego elementu tablicy), ukrywa za to szczegóły sposobu realizacji tego zamiaru (jak w instrukcji for, która mówi: „tworzę ten indeks, żeby wykorzystać go do wybierania kolejnych elementów tablicy"). Dlatego w tej książce wszędzie tam, gdzie to możliwe, będę stosował właśnie składnię foreach.

return

Niektóre słowa kluczowe języka reprezentują *bezwarunkowe rozgałęzienia* programu, czyli rozgałęzienia uniezależnione od wszelkich testów. Kategoria ta obejmuje słowa return, break, continue oraz pewien sposób wykonania skoku do etykietowanej instrukcji, podobny nieco do złowrogiego goto, znanego z innych języków programowania.

Słowo kluczowe return ma dwa zastosowania: ustala, jaka wartość zostanie zwrócona przez metodę (o ile typem zwracanym przez metodę nie jest void), oraz wymusza zakończenie wykonania metody ze zwróceniem tej wartości do wywołującego. Prezentowana wcześniej metoda test() mogłaby zostać przepisana tak, by z tego korzystała:

```
//: control/IfElse2.java
import static net.mindview.util.Print.*;

public class IfElse2 {
  static int test(int testval, int target) {
    if(testval > target)
      return +1;
    if(testval < target)
      return -1;
    return 0; // Równe
  }
  public static void main(String[] args) {
    print(test(10, 5));
    print(test(5, 10));
    print(test(5, 5));
  }
} /* Output:
1
-1
0
*///:~
```

Nie ma konieczności dopisywania else, ponieważ wykonanie metody i tak zostanie przerwane po wykonaniu return.

Jeśli metoda zadeklarowana jako niezwracająca wartości (z typem wartości zwracanej void) nie zawiera instrukcji return, kompilator zakłada obecność niejawnej instrukcji return tuż przed końcem ciała metody — nie trzeba więc wpisywać go tam jawnie. Jednak kiedy metoda zwraca jakiekolwiek wartości, trzeba samodzielnie zadbać o to, aby każda możliwa ścieżka wykonania metody kończyła się instrukcją return.

Ćwiczenie 6. Zmodyfikuj obie metody test() z dwóch poprzednich programów tak, aby przyjmowały dodatkowe argumenty, begin oraz end i aby wartość testval była sprawdzana pod kątem zawierania się w zakresie od begin do end (włącznie) (2).

break i continue

Wewnątrz każdej z instrukcji iteracji można sterować wykonaniem pętli, używając instrukcji break (przerwij) oraz continue (kontynuuj). Instrukcja break wychodzi z pętli, pomijając resztę jej instrukcji, a continue przerywa wykonanie aktualnej iteracji i wraca na początek pętli, rozpoczynając następną iterację.

Poniższy program pokazuje przykłady użycia break i continue wewnątrz pętli for i while.

```
//: control/BreakAndContinue.java
// Demonstracja działania słów kluczowych break i continue.
import static net.mindview.util.Range.*;

public class BreakAndContinue {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      if(i == 74) break; // Przerwanie pętli for
      if(i % 9 != 0) continue; // Następna iteracja
      System.out.print(i + " ");
    }
    System.out.println();
    // Z użyciem foreach:
    for(int i : range(100)) {
      if(i == 74) break; // Przerwanie pętli for
      if(i % 9 != 0) continue; // Następna iteracja
      System.out.print(i + " ");
    }
    System.out.println();
    int i = 0;
    // Pętla "nieskończona":
    while(true) {
      i++;
      int j = i * 27;
      if(j == 1269) break; // Przerwanie pętli
      if(i % 10 != 0) continue; // Na początek pętli
      System.out.print(i + " ");
    }
  }
} /* Output:
0 9 18 27 36 45 54 63 72
0 9 18 27 36 45 54 63 72
10 20 30 40
*///:~
```

W pętli for zmienna i nigdy nie osiągnie wartości 100, ponieważ instrukcja break przerywa pętlę, kiedy i jest równe 74. Normalnie używa się break w ten sposób, jeśli nie wiadomo, kiedy nastąpi warunek stopu. Instrukcja continue powoduje powrót na początek iteracji (a zatem zwiększenie i) za każdym razem, kiedy i nie jest podzielne przez 9. Kiedy to nastąpi, wypisywana jest wartość zmiennej i.

Druga pętla for ilustruje zastosowanie składni foreach; wynik wykonania pętli jest identyczny jak poprzednio.

Ostatnia pętla to pętla „nieskończona", która — teoretycznie — będzie wykonywana wiecznie. Jednak wewnątrz znajduje się instrukcja `break`, która spowoduje przerwanie powtarzania pętli. Widać także, że `continue` wraca na początek pętli, nie kończąc pozostałej części (dlatego w drugiej pętli wynik jest wypisywany tylko wtedy, gdy wartość `i` jest podzielna przez 10). W wynikach została wypisana wartość 0, ponieważ 0 % 9 daje 0.

Drugą postacią pętli nieskończonej jest `for(;;)`. Kompilator traktuje `while(true)` i `for(;;)` tak samo, więc to, którego używamy, jest kwestią gustu.

Ćwiczenie 7. Zmodyfikuj ćwiczenie 1. tak, aby program był przerywany instrukcją `break` przy wartości 99. Spróbuj też użyć tam słowa `return` (1).

Niesławne „goto"

Słowo kluczowe `goto` jest obecne w językach programowania od samego początku. Właściwie od `goto` zaczęło się sterowanie wykonaniem programu w językach asemblerowych: „Jeśli zachodzi warunek A, to skocz tutaj, jeśli nie zachodzi, to skocz tam". Czytając kod asemblerowy stworzony przez praktycznie dowolny kompilator, można zobaczyć, że zawiera on wiele skoków (kompilator Javy generuje własny „kod asemblerowy", który nie jest jednak wykonywany przez procesor komputera, lecz przez wirtualną maszynę Javy — ang. *Java Virtual Machine*).

Jednakże instrukcja `goto` jest skokiem na poziomie kodu źródłowego i to doprowadziło do jej niesławy. Jeśli program ma ciągle skakać z jednego miejsca w inne, to czy nie da się tak zreorganizować jego kodu, by jego przepływ sterowania nie był aż taki „skoczny"? Krytyka instrukcji `goto` rozpoczęła się od publikacji sławnego artykułu „Goto uznawane za szkodliwe" Edsgera Dijkstry i od tamtej pory nagonka na `goto` stała się popularną rozrywką, a zwolennicy wyklętego słowa kluczowego muszą ukrywać swoje poglądy.

Jak to często ma miejsce w podobnych sytuacjach, nie należy popadać w skrajności. Problemem nie jest używanie `goto`, ale nadużywanie `goto` — w niektórych sytuacjach `goto` jest tak naprawdę najlepszą metodą zorganizowania przepływu sterowania.

Mimo że `goto` jest w Javie słowem zarezerwowanym, to nie jest używane w języku; w Javie nie ma `goto`. Jednakże posiada ona coś, co wygląda podobnie jak skok i jest związane z instrukcjami `break` i `continue`. Nie jest to skok, ale raczej metoda na przerwanie iteracji. Powodem, dla którego metoda ta wiązana jest z dyskusjami o `goto`, jest używanie tego samego mechanizmu — etykiety.

Etykieta to identyfikator, po którym następuje dwukropek, jak poniżej:

```
etykieta1:
```

W Javie etykiety stosuje się *jedynie* przed instrukcją iteracji. I to tuż przed — pomiędzy etykietą i iteracją nie można wstawiać żadnych innych instrukcji. Wstawianie etykiety *przed* iteracją ma sens jedynie wtedy, gdy wewnątrz niej umieszczona jest kolejna iteracja lub instrukcja wyboru (`switch`). Słowa kluczowe `break` i `continue` normalnie powodują przerwanie bieżącej pętli, ale użyte z etykietą przerywają wszystkie pętle aż do poziomu, na którym znajduje się ta etykieta.

```
etykieta1:
iteracja-zewnętrzna {
  iteracja-wewnętrzna {
    // ...
    break; // (1)
    // ...
    continue; // (2)
    // ...
    continue etykieta1; // (3)
    // ...
    break etykieta1; // (4)
  }
}
```

W przypadku 1. `break` powoduje przerywanie wewnętrznej iteracji i znajdziemy się w iteracji zewnętrznej. W przypadku 2. `continue` powoduje przejście do początku iteracji wewnętrznej. Natomiast w przypadku 3. `continue etykieta1` przerywa wewnętrzną *oraz* zewnętrzną iterację, wracając do `etykieta1`. Następnie pętla jest kontynuowana, ale zaczynając od zewnętrznej iteracji. W przypadku 4. `break etykieta1` również przerywa wszystkie pętle do poziomu `etykieta1`, ale nie wchodzi ponownie w iterację. Przerywane są obydwie iteracje.

Oto przykład używający pętli `for`:

```
//: control/LabeledFor.java
// Pętle for z instrukcjami break i continue ze skokiem do etykiety.
import static net.mindview.util.Print.*;

public class LabeledFor {
  public static void main(String[] args) {
    int i = 0;
    outer: // Tu nie można wstawiać instrukcji
    for(; true ;) { // pętla nieskończona
      inner: // Tu nie można wstawiać instrukcji
      for(; i < 10; i++) {
        print("i = " + i);
        if(i == 2) {
          print("continue");
          continue;
        }
        if(i == 3) {
          print("break");
          i++; // Inaczej i nie doczekałoby
               // inkrementacji.
          break;
        }
        if(i == 7) {
          print("continue outer");
          i++; // Inaczej i nie doczekałoby
               // inkrementacji.
          continue outer;
        }
        if(i == 8) {
          print("break outer");
          break outer;
        }
```

```
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            print("continue inner");
            continue inner;
          }
        }
      }
    }
    // Nie można skoczyć do etykiet za pętlami
  }
} /* Output:
i = 0
continue inner
i = 1
continue inner
i = 2
continue
i = 3
break
i = 4
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer
*///:~
```

Można zauważyć, że break przerywa pętlę for, a wyrażenie inkrementujące wykonywane jest dopiero na końcu przejścia przez pętlę. Ponieważ break pomija wyrażenie inkrementacji, to w przypadku i==3 inkrementacja musi zostać wykonana bezpośrednio. Również instrukcja continue outer w przypadku i==7 wraca na początek pętli, pomijając inkrementację, dlatego tu również konieczna jest bezpośrednia inkrementacja.

Gdyby nie instrukcja break outer, nie byłoby sposobu, żeby znajdując się w pętli wewnętrznej, wyjść z pętli zewnętrznej, ponieważ samo break może przerwać tylko najgłębszą pętlę (to samo odnosi się do continue).

Oczywiście w przypadku, kiedy przerwanie pętli spowoduje jednocześnie wyjście z metody, można po prostu użyć return.

Oto demonstracja etykietowanych instrukcji break i continue użytych z pętlą while:

```
//: control/LabeledWhile.java
// Pętle while z instrukcjami break i continue ze skokami do etykiet.
import static net.mindview.util.Print.*;

public class LabeledWhile {
  public static void main(String[] args) {
    int i = 0;
    outer:
    while(true) {
      print("Zewnętrzna pętla while");
      while(true) {
```

```
        i++;
        print("i = " + i);
        if(i == 1) {
          print("continue (zewnętrzna)");
          continue;
        }
        if(i == 3) {
          print("continue (zewnętrzna)");
          continue outer;
        }
        if(i == 5) {
          print("break");
          break;
        }
        if(i == 7) {
          print("break (zewnętrzna)");
          break outer;
        }
      }
    }
  }
} /* Output:
Zewnętrzna pętla while
i = 1
continue
i = 2
i = 3
continue (zewnętrzna)
Zewnętrzna pętla while
i = 4
i = 5
break
Zewnętrzna pętla while
i = 6
i = 7
break (zewnętrzna)
*///:~
```

Te same zasady odnoszą się do `while`:

1. Zwykłe `continue` wraca na początek najbardziej wewnętrznej pętli i kontynuuje wykonanie.
2. Etykietowane `continue` idzie do etykiety i ponownie wchodzi do pętli zaraz za tą etykietą.
3. Zwykłe `break` wychodzi z pętli.
4. Etykietowane `break` przechodzi poza koniec pętli oznaczonej etykietą.

Ważne jest, aby zapamiętać, że w Javie *jedynym* powodem użycia etykiet są zagnieżdżone pętle, w których trzeba użyć `break` lub `continue` przechodzących przez więcej niż jeden poziom zagnieżdżenia.

W swoim artykule „Goto uznawane za szkodliwe" Dijkstra szczególnie sprzeciwiał się etykietom, nie skokom. Zauważył, że liczba błędów zwiększała się proporcjonalnie do liczby etykiet w programie, a etykiety i skoki powodują, że trudno analizować programy.

Jednak nie dotyczy to etykiet w Javie, ponieważ ich umieszczanie jest ograniczone i nie można wykonywać skoków ad hoc. Równie interesujące jest spostrzeżenie, że w tym przypadku właściwość języka stała się bardziej użyteczna przez ograniczenie możliwości instrukcji.

switch

Instrukcja `switch` jest czasem nazywana *instrukcją wielokrotnego wyboru*. Instrukcja ta na podstawie wyrażenia całkowitego wybiera odpowiedni fragmentu kodu. Ma postać:

```
switch (selektor-całkowity) {
  case wartość-całkowita1 : instrukcja; break;
  case wartość-całkowita2 : instrukcja; break;
  case wartość-całkowita3 : instrukcja; break;
  case wartość-całkowita4 : instrukcja; break;
  case wartość-całkowita5 : instrukcja; break;
        // ...
  default: instrukcja;
}
```

`Selektor-całkowity` jest wyrażeniem zwracającym wartość całkowitą. Instrukcja `switch` porównuje wynik tego wyrażenia z każdą ze stałych `wartość-całkowita`. Jeśli znajdzie przypadek (ang. *case*) zgodny z wartością wyrażenia, to wykonywana jest odpowiadająca mu `instrukcja` (prosta lub złożona). Jeśli żaden przypadek nie jest zgodny z wartością wyrażenia, wykonywana jest instrukcja związana z przypadkiem `default` (ang. *domyślny*).

Z powyższej definicji wynika, że każdy przypadek kończy się instrukcją `break`, która powoduje, że wykonanie jest kontynuowane od pierwszej instrukcji za `switch`. Jest to wygodny sposób budowania instrukcji `switch`, jednak `break` jest opcjonalny. Jeśli go nie ma, to wykonywany jest kod dla kolejnych przypadków, aż do napotkania `break`. Mimo że najczęściej takie zachowanie nie jest pożądane, może ono być wykorzystane przez doświadczonego programistę. Zauważ, że ostatnia instrukcja następująca po `default` nie kończy się `break`, ponieważ sterowanie i tak przechodzi do tego samego miejsca, do którego zostałoby przeniesione przez `break`. Bez żadnej szkody można wstawić `break` na koniec wyrażenia `default`, jeśli ktoś uważa, że tak jest bardziej elegancko.

Instrukcja `switch` jest prostą metodą zapisu wielokrotnego wyboru (tzn. wyboru z wielu ścieżek wykonania), ale wymaga selektora zwracającego wartości całkowite, tak jak `int` lub `char`. Nie można użyć jako selektora na przykład łańcucha znakowego lub liczby zmiennoprzecinkowej. Dla typów niecałkowitych trzeba używać serii instrukcji `if`. Pod koniec następnego rozdziału dowiesz się, że Java SE5 łagodzi to ograniczenie za pomocą wyliczeń `enum`, które świetnie nadają się do stosowania z instrukcją `switch`.

W poniższym przykładzie użyto instrukcji `switch` do określenia, czy wylosowana litera jest spółgłoską czy samogłoską:

```
//: control/VowelsAndConsonants.java
// Ilustracja instrukcji wielokrotnego wyboru.
import java.util.*;
import static net.mindview.util.Print.*;
```

```
public class VowelsAndConsonants {
  public static void main(String[] args) {
    Random rand = new Random(47);
    for(int i = 0; i < 100; i++) {
      int c = rand.nextInt(26) + 'a';
      printnb((char)c + ", " + c + ": ");
      switch(c) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u': print("samogłoska");
                  break;
        case 'y':
        case 'w': print("czasam samogłoska");
                  break;
        default:  print("spółgłoska");
      }
    }
  }
} /* Output:
y, 121: czasam samogłoska
n, 110: spółgłoska
z, 122: spółgłoska
b, 98: spółgłoska
r, 114: spółgłoska
n, 110: spółgłoska
y, 121: czasam samogłoska
g, 103: spółgłoska
c, 99: spółgłoska
f, 102: spółgłoska
o, 111: samogłoska
w, 119: czasam samogłoska
z, 122: spółgłoska
...
*///:~
```

Ponieważ metoda `Random.nextInt(26)` zwraca wartość pomiędzy 0 a 25, wystarczy dodać do niej przesunięcie kodu litery 'a', aby rzutować generowany zakres na zakres kodów małych liter alfabetu. Występujące w pojedynczych cudzysłowach znaki przy instrukcjach `case` reprezentują takie właśnie wartości całkowite, przez co nadają się do porównywania.

Warto również zwrócić uwagę, w jaki sposób można ułożyć instrukcje `case` „jedną na drugiej", aby umożliwić wielokrotne wybranie tego samego fragmentu kodu. Należy pamiętać o umieszczeniu instrukcji `break` na końcu każdego przypadku, gdyż w przeciwnym razie wykonany zostanie kod następnego przypadku.

W instrukcji:

```
int c = rand.nextInt(26) + 'a';
```

wywołanie metody `Random.nextInt()` zwraca wartość typu `int` z zakresu od 0 do 25, dodawaną następnie do wartości 'a'. Oznacza to automatyczną konwersję 'a' na wartość typu `int` dla potrzeb operacji dodawania.

Aby wypisać `c` jako znak, należy rzutować `c` na typ `char`; w innym przypadku na wyjściu pojawi się wartość liczbowa, czyli kod znaku.

Ćwiczenie 8. Utwórz instrukcję `switch`, która będzie dla każdego przypadku wypisywać komunikat na wyjściu; umieść utworzoną instrukcję wyboru wewnątrz pętli `for` tak, aby wypróbować każdy przypadek. Każdy przypadek zakończ instrukcją `break` i sprawdź działanie programu; potem usuń instrukcje `break` i zobacz, jak się ono zmieni (2).

Ćwiczenie 9. *Ciąg Fibonacciego* to ciąg liczb 1, 1, 2, 3, 5, 8, 13, 21, 34 i tak dalej; każdy kolejny wyraz ciągu (począwszy od trzeciego) jest obliczany jako suma dwóch poprzednich. Napisz metodę, która za pośrednictwem argumentu przyjmie liczbę całkowitą i wypisze zadaną nią liczbę wyrazów ciągu Fibonacciego (od początku, a więc np. uruchomienie programu `java Fibonacci 5` (gdzie `Fibonacci` to nazwa klasy z metodą `main()`) powinno zaowocować wypisaniem: 1, 1, 2, 3, 5) (4).

Ćwiczenie 10. *Liczba wampirza* posiada parzystą liczbę cyfr, a tworzy się ją, mnożąc pary liczb zawierające po połowie cyfr wyniku. Cyfry mogą być wybierane z pierwotnej liczby w dowolnej kolejności. Nie dopuszcza się w liczbie par zer na końcu liczby. Oto przykłady:

1260 = 21 * 60

1827 = 21 * 87

2187 = 27 * 81

Napisz program, który wyszuka wszystkie czterocyfrowe liczby wampirze (ćwiczenie według pomysłu Dana Forhana) (5).

Podsumowanie

Rozdział ten kończy przegląd podstawowych właściwości większości języków programowania: wykonywanie obliczeń, kolejność operatorów, rzutowanie typów oraz instrukcje wyboru i iteracji. Czytelnik jest już gotów na spotkanie ze światem programowania zorientowanego obiektowo. W następnym rozdziale zostaną przedstawione ważne kwestie związane z inicjalizacją i usuwaniem obiektów, po czym — w kolejnym rozdziale — omówione zostanie pojęcie ukrywania implementacji.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 5.

Inicjalizacja i sprzątanie

W miarę postępu rewolucji komputerowej „niebezpieczne" programowanie stało się jedną z głównych przyczyn, które uczyniły programowanie bardzo drogim.

Dwie kwestie bezpieczeństwa to *inicjalizacja* i *sprzątanie*. W języku C powstaje wiele błędów, ponieważ programiści zapominają o zainicjalizowaniu zmiennej. Szczególnie wyraźnie objawia się to w przypadku bibliotek: użytkownicy nie wiedzą, jak inicjować komponent biblioteczny, czy też, że w ogóle muszą to zrobić. Sprzątanie jest problemem szczególnym, gdyż łatwo zapomnieć o elemencie, którego już się nie używa. Tak więc zasoby wykorzystywane przez taki element są zatrzymane i można łatwo doprowadzić do sytuacji, gdy ich zabraknie (przeważnie dotyczy to pamięci).

C++ wprowadza pojęcie *konstruktora*, czyli specjalnej metody automatycznie wywoływanej podczas tworzenia obiektu. Java także zaadaptowała konstruktory, ale dodatkowo posiada odśmiecacz pamięci (ang. *garbage collector*), który automatycznie zwalnia zasoby pamięci, gdy nie są już wykorzystywane. Rozdział ten opisuje zagadnienia inicjalizacji i sprzątania oraz ich działanie w Javie.

Gwarantowana inicjalizacja przez konstruktor

Można sobie wyobrazić metodę o nazwie `initialize()` dla każdej napisanej przez nas klasy. Jej nazwa jest wskazówką mówiącą, że powinna być ona wywołana przed pierwszym skorzystaniem z obiektu. Niestety, oznacza to, że użytkownik musi pamiętać o jej wywołaniu. W Javie projektant klasy może zagwarantować inicjalizację każdego obiektu poprzez dostarczenie specjalnej metody zwanej *konstruktorem*. Jeśli klasa posiada konstruktor, to zostanie on automatycznie wywołany podczas tworzenia obiektu, zanim użytkownik uzyska do niego dostęp. A zatem inicjalizacja jest zapewniona.

Kolejnym wyzwaniem jest wybranie nazwy dla tej metody. Są dwa sposoby. Po pierwsze, każda użyta przez nas nazwa może wejść w konflikt z nazwą, jakiej — być może — chcielibyśmy użyć dla składowej klasy. Po drugie, ponieważ kompilator jest odpowiedzialny za wywołanie konstruktora, to musi zawsze wiedzieć, którą metodę ma wywołać.

Rozwiązanie C++ wydaje się najprostsze i najbardziej logiczne, stąd zostało także wykorzystane w Javie — nazwa konstruktora jest taka sama, jak nazwa klasy. Wydaje się sensowne, że taka metoda zostanie wywołana automatycznie podczas inicjalizacji.

Oto prosta klasa z konstruktorem:

```
//: initialization/SimpleConstructor.java
// Demonstracja prostego konstruktora.

class Rock {
  Rock() { // To jest konstruktor
    System.out.print("Kamień ");
  }
}

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock();
  }
} /* Output:
Kamień Kamień Kamień Kamień Kamień Kamień Kamień Kamień Kamień Kamień
*///:~
```

Teraz, gdy obiekt jest tworzony:

```
new Rock();
```

rezerwowana jest pamięć i wywoływany jest konstruktor. Gwarantowane jest więc poprawne zainicjalizowanie obiektu, zanim uzyskamy do niego dostęp.

Zauważ, że styl kodowania na podstawie pisania pierwszego znaku wszystkich metod małą literą nie dotyczy konstruktorów, ponieważ nazwa konstruktora musi *dokładnie* pasować do nazwy klasy.

Konstruktor nieprzyjmujący żadnych argumentów zwie się *konstruktorem domyślnym*. W dokumentacji Javy występuje zwykle jako *konstruktor bezargumentowy* (ang. *no-args constructor*), ale termin „konstuktor domyślny" na dobre wszedł do słownika programistów na długo przed pojawieniem się Javy, więc i ja będę go używał. Podobnie jak dowolna metoda klasy, konstruktor może mieć argumenty pozwalające określić, *jak* obiekt ma zostać stworzony. Powyższy przykład można łatwo zmienić tak, aby konstruktor pobierał argument:

```
//: initialization/SimpleConstructor2.java
// Konstruktory mogą przyjmować argumenty.

class Rock2 {
  Rock2(int i) {
    System.out.print("Kamień " + i + " ");
  }
}

public class SimpleConstructor2 {
  public static void main(String[] args) {
    for(int i = 0; i < 8; i++)
      new Rock2(i);
```

```
  }
} /* Output:
Kamień 0 Kamień 1 Kamień 2 Kamień 3 Kamień 4 Kamień 5 Kamień 6 Kamień 7
*///:~
```

Argumenty konstruktora zapewniają możliwości parametryzacji inicjalizacji obiektu. Na przykład, jeżeli klasa `Tree`, reprezentująca drzewo, posiada konstruktor pobierający jeden argument będący liczbą całkowitą określającą wysokość drzewa, to stworzylibyśmy obiekt `Tree` w następujący sposób:

```
Tree t = new Tree(4);  // drzewo 4-metrowe
```

Jeżeli `Tree(int)` jest jedynym konstruktorem, to kompilator nie pozwoli stworzyć obiektu `Tree` w żaden inny sposób.

Konstruktory eliminują wiele problemów i czynią kod bardziej czytelnym. W poprzednim przykładzie nie widać jawnego wywołania jakiejś metody `initialize()`, która byłaby pojęciowo odseparowana od tworzenia obiektu. W Javie pojęcia tworzenia i inicjalizacji połączono — nie można mieć jednego bez drugiego.

Konstruktor jest specjalnym rodzajem metody, ponieważ nie posiada wartości zwracanej. Jest to wyraźnie coś innego niż zwracanie wartości typu `void`, kiedy metoda niczego nie zwraca, ale ciągle mamy możliwość zmiany tej sytuacji. Konstruktory niczego nie zwracają i nie można tego zmienić (wyrażenie `new` zwraca referencję do nowo utworzonego obiektu, jednak sam konstruktor nie zwraca żadnej wartości). Gdyby istniała możliwość zwracania wartości i gdybyśmy mogli wybrać tę wartość, kompilator musiałby wiedzieć, co z tą wartością zrobić.

Ćwiczenie 1. Utwórz klasę zawierającą niezainicjalizowaną referencję klasy `String`. Wykaż, że ta referencja zostanie zainicjalizowana przez system wykonawczy wartością pustą (`null`) (1).

Ćwiczenie 2. Utwórz klasę ze składową `String` inicjalizowaną w miejscu definicji i drugą klasę z taką samą składową inicjalizowaną z poziomu konstruktora. Czym różnią się obie inicjalizacje (2)?

Przeciążanie metod

Jedną z ważnych cech każdego języka programowania jest używanie nazw. Kiedy tworzymy obiekt, nadajemy nazwę fragmentowi pamięci. Metoda to nazwa pewnego działania. Do wszystkich obiektów i metod odwołujemy się, wykorzystując ich nazwy. Odpowiednie ich dobranie ułatwia zarówno nam, jak i innym zrozumienie kodu.

Problem pojawia się, gdy odwzorujemy pojęcia języka naturalnego na język programowania. Często to samo słowo posiada wiele różnych znaczeń — jest *przeciążone*. Jest to użyteczne szczególnie wtedy, gdy różnice są błahe. Mówimy: „umyj samochód", „umyj psa". Byłoby to głupie, gdybyśmy mówili „samochódUmyj samochód" i „psaUmyj psa", gdyż słuchacz nie musi czynić żadnego odróżnienia wykonywanej czynności. Większość

języków naturalnych jest redundantna, toteż nawet jeżeli opuścimy klika słów, wciąż można zrozumieć znaczenie. Nie potrzebujemy unikatowych identyfikatorów — możemy wydedukować znaczenie na podstawie kontekstu.

Większość języków programowania (język C w szczególności) wymaga posiadania unikatowych identyfikatorów dla każdej *funkcji*. Stąd nic można mieć jednej funkcji o nazwie `print()` do wypisywania liczb całkowitych i innej `print()` do wypisywania liczb zmiennoprzecinkowych — nazwa każdej funkcji musi być unikatowa.

W Javie (a także C++) inny czynnik wymusza przeciążanie nazw metod — jest nim konstruktor. Ponieważ nazwa konstruktora jest zdeterminowana wcześniej przez nazwę klasy, może istnieć tylko jedna taka nazwa. Co jednak zrobić, jeśli chcemy tworzyć obiekt na więcej niż jeden sposób? Na przykład, przypuśćmy, że budujemy klasę, która może być inicjalizowana w sposób standardowy albo przez odczyt informacji z pliku. Potrzebujemy dwóch konstruktorów: domyślnego i takiego, który jako argument będzie pobierał obiekt `String`, będący nazwą pliku zawierającego dane potrzebne do inicjalizacji obiektu. Oba są konstruktorami, a więc muszą mieć taką samą nazwę — będącą nazwą klasy. Tak więc *przeciążenie metod* jest niezbędne, jeśli chcemy wykorzystywać tę samą nazwę metody z różnymi typami argumentów. Mimo iż przeciążenie jest koniecznością w przypadku konstruktorów, udogodnienie przez nie zapewniane ma charakter ogólny i może być używane w stosunku do każdej metody.

Poniższy przykład pokazuje oba przypadki przeciążenia — przeciążenie konstruktora i zwykłej metody:

```
//: initialization/Overloading.java
// Demonstracja przeciążania konstruktorów
// i zwykłych metod klas.
import static net.mindview.util.Print.*;

class Tree {
  int height;
  Tree() {
    print("Sadzenie sadzonki");
    height = 0;
  }
  Tree(int initialHeight) {
    height = initialHeight;
    print("Tworzenie nowego drzewka, wysokiego na " +
      height + " metr(y)(ów)");
  }
  void info() {
    print("Drzewo ma " + height + " metr(y)(ów) wysokości");
  }
  void info(String s) {
    print(s + ": drzewo ma " + height + " metr(y)(ów) wysokości");
  }
}

public class Overloading {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Tree t = new Tree(i);
```

```
      t.info();
      t.info("metoda przeciążona");
    }
    // Przeciążony konstruktor:
    new Tree();
  }
} /* Output:
Tworzenie nowego drzewka, wysokiego na 0 metr(y)(ów)
Drzewo ma 0 metr(y)(ów) wysokości
metoda przeciążona: drzewo ma 0 metr(y)(ów) wysokości
Tworzenie nowego drzewka, wysokiego na 1 metr(y)(ów)
Drzewo ma 1 metr(y)(ów) wysokości
metoda przeciążona: drzewo ma 1 metr(y)(ów) wysokości
Tworzenie nowego drzewka, wysokiego na 2 metr(y)(ów)
Drzewo ma 2 metr(y)(ów) wysokości
metoda przeciążona: drzewo ma 2 metr(y)(ów) wysokości
Tworzenie nowego drzewka, wysokiego na 3 metr(y)(ów)
Drzewo ma 3 metr(y)(ów) wysokości
metoda przeciążona: drzewo ma 3 metr(y)(ów) wysokości
Tworzenie nowego drzewka, wysokiego na 4 metr(y)(ów)
Drzewo ma 4 metr(y)(ów) wysokości
metoda przeciążona: drzewo ma 4 metr(y)(ów) wysokości
Sadzenie sadzonki
*///:~
```

Obiekt Tree może zostać stworzony albo poprzez zasadzenie, albo jako roślina dorastająca w szkółce, posiadająca już pewną wysokość. Aby to umożliwić, istnieją dwa konstruktory: konstruktor domyślny oraz konstruktor pobierający dotychczasową wysokość.

Można również chcieć wywołać metodę `info()` na więcej niż jeden sposób. Na przykład, jeśli mamy dodatkowy komunikat, który chcemy wyświetlić, możemy użyć wywołania `info(String)`, jeśli natomiast nie mamy niczego więcej do powiedzenia, to możemy posłużyć się wywołaniem `info()`. Dziwnie wyglądałoby nadawanie dwóch oddzielnych nazw czemuś, co w sposób oczywisty jest tym samym pojęciem. Na szczęście przeciążanie metod pozwala na zastosowanie tej samej nazwy w obu przypadkach.

Rozróżnianie przeciążonych metod

Jeśli metody mają tę samą nazwę, to skąd Java wie, którą metodę mamy na myśli? Reguła jest prosta: każda metoda przeciążona musi określać unikatową listę typów argumentów.

Jeśli się przez chwilę nad tym zastanowić, ma to sens: jak inaczej programista mógłby określić różnicę pomiędzy dwoma metodami o tej samej nazwie, jeśli nie przez typy jej argumentów?

Nawet różnica w kolejności argumentów jest wystarczająca, aby odróżnić dwie metody (chociaż normalnie nie będziemy stosować takiego rozwiązania, ponieważ powoduje ono powstawanie kodu trudnego w utrzymaniu).

```
//: initialization/OverloadingOrder.java
// Przeciążanie przez zmianę kolejności argumentów.
import static net.mindview.util.Print.*;

public class OverloadingOrder {
  static void f(String s, int i) {
    print("String: " + s + ", int: " + i);
```

```
  }
  static void f(int i, String s) {
    print("int: " + i + ", String: " + s);
  }
  public static void main(String[] args) {
    f("String pierwszy", 11);
    f(99, "Int pierwszy");
  }
} /* Output:
String: String pierwszy, int: 11
int: 99, String: Int pierwszy
*///:~
```

Dwie metody `f()` posiadają identyczne argumenty, jednakże ich kolejność jest inna i dzięki temu są odmienne.

Przeciążanie a typy podstawowe

Typy podstawowe mogą być automatycznie promowane z typów mniejszych do większych, co w połączeniu z przeciążaniem może wywołać małe zamieszanie. Następujący przykład pokazuje, co się dzieje, kiedy przekazywany jest typ podstawowy do metody przeciążonej:

```
//: initialization/PrimitiveOverloading.java
// Promocja typów podstawowych a przeciążanie.
import static net.mindview.util.Print.*;

public class PrimitiveOverloading {
  void f1(char x) { printnb("f1(char) "); }
  void f1(byte x) { printnb("f1(byte) "); }
  void f1(short x) { printnb("f1(short) "); }
  void f1(int x) { printnb("f1(int) "); }
  void f1(long x) { printnb("f1(long) "); }
  void f1(float x) { printnb("f1(float) "); }
  void f1(double x) { printnb("f1(double) "); }

  void f2(byte x) { printnb("f2(byte) "); }
  void f2(short x) { printnb("f2(short) "); }
  void f2(int x) { printnb("f2(int) "); }
  void f2(long x) { printnb("f2(long) "); }
  void f2(float x) { printnb("f2(float) "); }
  void f2(double x) { printnb("f2(double) "); }

  void f3(short x) { printnb("f3(short) "); }
  void f3(int x) { printnb("f3(int) "); }
  void f3(long x) { printnb("f3(long) "); }
  void f3(float x) { printnb("f3(float) "); }
  void f3(double x) { printnb("f3(double) "); }

  void f4(int x) { printnb("f4(int) "); }
  void f4(long x) { printnb("f4(long) "); }
  void f4(float x) { printnb("f4(float) "); }
  void f4(double x) { printnb("f4(double) "); }

  void f5(long x) { printnb("f5(long) "); }
  void f5(float x) { printnb("f5(float) "); }
  void f5(double x) { printnb("f5(double) "); }
```

```
  void f6(float x) { printnb("f6(float) "); }
  void f6(double x) { printnb("f6(double) "); }

  void f7(double x) { printnb("f7(double) "); }

  void testConstVal() {
    printnb("5: ");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5); print();
  }
  void testChar() {
    char x = 'x';
    printnb("char: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testByte() {
    byte x = 0;
    printnb("byte: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testShort() {
    short x = 0;
    printnb("short: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testInt() {
    int x = 0;
    printnb("int: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testLong() {
    long x = 0;
    printnb("long: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testFloat() {
    float x = 0;
    printnb("float: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  void testDouble() {
    double x = 0;
    printnb("double: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
  }
  public static void main(String[] args) {
    PrimitiveOverloading p =
      new PrimitiveOverloading();
    p.testConstVal();
    p.testChar();
    p.testByte();
    p.testShort();
    p.testInt();
    p.testLong();
    p.testFloat();
    p.testDouble();
  }
} /* Output:
```

```
5: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
char: f1(char) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
byte: f1(byte) f2(byte) f3(short) f4(int) f5(long) f6(float) f7(double)
short: f1(short) f2(short) f3(short) f4(int) f5(long) f6(float) f7(double)
int: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
long: f1(long) f2(long) f3(long) f4(long) f5(long) f6(float) f7(double)
float: f1(float) f2(float) f3(float) f4(float) f5(float) f6(float) f7(double)
double: f1(double) f2(double) f3(double) f4(double) f5(double) f6(double) f7(double)
*///:~
```

Gdy przyjrzymy się wynikowi działania tego programu, to widać, że stała wartość 5 jest traktowana jako wartość typu `int`, a zatem jeśli istnieje dostępna wersja przeciążonej metody pobierająca argument tego typu, wtedy jest ona wykorzystywana. We wszystkich innych przypadkach, kiedy mamy typ danych, który jest mniejszy niż argument metody, to typ ten jest promowany. Typ `char` zachowuje się odrobinę inaczej, ponieważ w przypadku braku metody dla dokładnie takiego typu będzie promowany do typu `int`.

Co się stanie, jeżeli argument jest *większy* niż typ argumentu oczekiwanego przez metodę przeciążoną? Modyfikacja powyższego programu daje odpowiedź:

```
//: c04:Demotion.java
//: initialization/Demotion.java
// Degradacja typów podstawowych a przeciążanie.
import static net.mindview.util.Print.*;

public class Demotion {
  void f1(char x) { print("f1(char)"); }
  void f1(byte x) { print("f1(byte)"); }
  void f1(short x) { print("f1(short)"); }
  void f1(int x) { print("f1(int)"); }
  void f1(long x) { print("f1(long)"); }
  void f1(float x) { print("f1(float)"); }
  void f1(double x) { print("f1(double)"); }

  void f2(char x) { print("f2(char)"); }
  void f2(byte x) { print("f2(byte)"); }
  void f2(short x) { print("f2(short)"); }
  void f2(int x) { print("f2(int)"); }
  void f2(long x) { print("f2(long)"); }
  void f2(float x) { print("f2(float)"); }

  void f3(char x) { print("f3(char)"); }
  void f3(byte x) { print("f3(byte)"); }
  void f3(short x) { print("f3(short)"); }
  void f3(int x) { print("f3(int)"); }
  void f3(long x) { print("f3(long)"); }

  void f4(char x) { print("f4(char)"); }
  void f4(byte x) { print("f4(byte)"); }
  void f4(short x) { print("f4(short)"); }
  void f4(int x) { print("f4(int)"); }

  void f5(char x) { print("f5(char)"); }
  void f5(byte x) { print("f5(byte)"); }
  void f5(short x) { print("f5(short)"); }
```

```
  void f6(char x) { print("f6(char)"); }
  void f6(byte x) { print("f6(byte)"); }
  void f7(char x) { print("f7(char)"); }

  void testDouble() {
    double x = 0;
    print("Argument typu double:");
    f1(x);f2((float)x);f3((long)x);f4((int)x);
    f5((short)x);f6((byte)x);f7((char)x);
  }
  public static void main(String[] args) {
    Demotion p = new Demotion();
    p.testDouble();
  }
} /* Output:
Argument typu double:
f1(double)
f2(float)
f3(long)
f4(int)
f5(short)
f6(byte)
f7(char)
*///:~
```

Tym razem metody pobierają argumenty mniejszych typów podstawowych. Jeżeli argument jest typu większego, należy go *zrzutować* do odpowiedniego typu, stosując nazwę typu w nawiasach. Jeśli się tego nie zrobi, to kompilator wyświetli komunikat o błędzie.

Przeciążanie przez wartości zwracane

Często dziwimy się: „Dlaczego tylko nazwy klas i lista argumentów metody? Dlaczego nie rozróżniać metod na podstawie wartości zwracanych?". Na przykład dwie poniższe metody o tej samej nazwie i argumentach są łatwo rozróżnialne:

```
void f() {}
int f() { return 1; }
```

Takie rozwiązanie działa dobrze, dopóki kompilator potrafi w sposób jednoznaczny określić znaczenie na podstawie kontekstu, jak w wyrażeniu `int x = f()`. Można jednak wywołać metodę, ignorując wartość zwracaną, co jest często przedstawiane jako *wywołanie metody dla jej efektu ubocznego*, ponieważ nie dbamy o wartość zwracaną, a zamiast tego chcemy uzyskać inne efekty związane z wywołaniem metody. Zatem jeśli wywołamy metodę w sposób następujący:

```
f();
```

to jak Java może określić, która metoda `f()` powinna być wywołana? No i jak ktoś mógłby odczytać taki kod, widząc ten zapis? Z tego powodu nie można użyć typu wartości zwracanej do odróżnienia metod przeciążonych.

Konstruktory domyślne

Jak wspominałem wcześniej, konstruktor domyślny („bezargumentowy") to konstruktor niepobierający żadnych argumentów, używany do tworzenia „prostego" obiektu. Jeśli napiszemy klasę bez konstruktora, to kompilator automatycznie utworzy konstruktor domyślny za nas, na przykład:

```
//: initialization/DefaultConstructor.java

class Bird {}

public class DefaultConstructor {
  public static void main(String[] args) {
    Bird b = new Bird(); // Domyślny!
  }
} ///:~
```

Wiersz:

```
new Bird();
```

tworzy nowy obiekt i wywołuje jego konstruktor domyślny, nawet jeżeli żaden nie został jawnie zdefiniowany. Bez niego nie byłoby metody, której wywołanie tworzyłoby obiekt. Jednak jeśli zdefiniujemy jakikolwiek konstruktor (z argumentami bądź bez), kompilator *nie* stworzy żadnego samodzielnie:

```
//: initialization/NoSynthesis.java

class Bird2 {
  Bird2(int i) {}
  Bird2(double d) {}
}

public class NoSynthesis {
  public static void main(String[] args) {
    //! Bird2 b = new Bird2();  // Brak domyślnego
    Bird2 b2 = new Bird2(1);
    Bird2 b3 = new Bird2(1.0);
  }
} ///:~
```

Jeżeli teraz wywołamy:

```
new Bird2();
```

kompilator poinformuje nas, że nie może odnaleźć odpowiedniego konstruktora. To tak, jakby w przypadku, gdy nie zdefiniujemy żadnego konstruktora, kompilator mówił: „Jesteś zobowiązany dostarczyć *jakiś* konstruktor, a więc pozwól, że zrobię jeden za Ciebie". Ale jeżeli napiszemy konstruktor, to kompilator uzna: „Napisałeś konstruktor, a więc wiesz, co robisz; jeśli nie zamieściłeś konstruktora domyślnego, to pewnie dlatego, że umyślnie chciałeś go wykluczyć".

Ćwiczenie 3. Utwórz klasę z konstruktorem domyślnym (nieprzyjmującym argumentów), który wypisze komunikat. Utwórz obiekt takiej klasy (1).

Ćwiczenie 4. Dodaj do klasy z poprzedniego ćwiczenia konstruktor przeciążony, który będzie przyjmował obiekt `String` i wypisywał jego zawartość razem z pierwotnym komunikatem (1).

Ćwiczenie 5. Utwórz klasę o nazwie `Dog` (pies) z przeciążoną metodą `bark()` (szczek). Metoda ta powinna zostać przeciążona dla różnych podstawowych typów danych i wypisywać różne rodzaje szczekania i wycia, zależnie od wersji metody. Napisz metodę `main()` wywołującą różne wersje przeciążone (2).

Ćwiczenie 6. Zmień poprzedni przykład tak, aby dwie z przeciążonych wersji metody `bark()` przyjmowały po dwa argumenty (różnych typów), ale w odwrotnej kolejności. Sprawdź, czy takie przeciążanie działa (1).

Ćwiczenie 7. Napisz klasę bez konstruktora, a potem w metodzie `main()` utwórz obiekt tej klasy, testując w ten sposób automatyczne tworzenie konstruktora domyślnego (1).

Słowo kluczowe this

Mając dwa obiekty tego samego typu, nazwane `a` i `b`, można się zastanowić, dlaczego można wywołać metodę `peel()` („obierz") dla obu tych obiektów:

```
//: initialization/BananaPeel.java

class Banana { void peel(int i) { /* ... */ } }

public class BananaPeel {
  public static void main(String[] args) {
    Banana a = new Banana(),
           b = new Banana();
    a.peel(1);
    b.peel(2);
  }
} ///:~
```

Skoro istnieje tylko jedna metoda o nazwie `peel()`, to skąd ona wie, czy jest wywoływana na rzecz obiektu `a` czy `b`?

Aby pozwolić na zapis kodu w wygodnej, obiektowo zorientowanej składni, w której „przesyła się komunikaty do obiektu", kompilator wykonuje za nas pewną ukrytą pracę. Istnieje bowiem tajemniczy pierwszy argument przekazywany do metody `peel()`, będący referencją do obiektu, na którym działamy. Tak więc dwa powyższe wywołania metod stają się czymś w rodzaju:

```
Banana.peel(a, 1);
Banana.peel(b, 2);
```

Wszystko odbywa się w sposób całkowicie niewidoczny, nie można zapisać tych wyrażeń i oczekiwać, że kompilator je zaakceptuje, jednak daje to dobre wyobrażenie o tym, co się dzieje.

Przypuśćmy, że znajdujemy się wewnątrz metody i chcemy uzyskać referencję do aktualnego obiektu. Ponieważ referencja ta jest przekazywana *potajemnie* przez kompilator, żaden identyfikator dla niej nie istnieje. Jednak do tego celu służy słowo kluczowe `this`. Słowo kluczowe `this` — które może być stosowane tylko wewnątrz metody niestatycznej — zwraca referencję do obiektu, na rzecz którego metoda została wywołana. Można traktować tę referencję jak każdą inną referencję do obiektu. Należy pamiętać, że wywołując metodę klasy z wnętrza innej metody tej klasy, nie ma potrzeby używania `this` — po prostu wywołuje się metodę. Aktualna referencja `this` jest automatycznie stosowana dla drugiej metody. Zatem można napisać:

```
//: initialization/Apricot.java
public class Apricot {
  void pick() { /* ... */ }
  void pit() { pick(); /* ... */ }
} ///:~
```

Wewnątrz `pit()` można by napisać `this.pick()`, ale nie ma takiej potrzeby[1]. Kompilator zrobi to za nas automatycznie. Słowo kluczowe `this` jest wykorzystywane tylko w tych specjalnych przypadkach, w których trzeba jawnie użyć referencji do aktualnego obiektu. Na przykład jest ono często używane w instrukcji `return`, kiedy chcemy zwrócić referencję do aktualnego obiektu:

```
//: initialization/Leaf.java
// Proste zastosowanie słowa "this".

public class Leaf {
  int i = 0;
  Leaf increment() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Leaf x = new Leaf();
    x.increment().increment().increment().print();
  }
} /* Output:
i = 3
*///:~
```

Ponieważ `increment()` zwraca referencję do aktualnego obiektu poprzez słowo kluczowe `this`, to z łatwością można przeprowadzić kilka operacji na tym samym obiekcie.

[1] Niektórzy w obsesyjny sposób zapisują słowo kluczowe `this` przed każdym wywołaniem metody i odwołaniem do pola, argumentując, że dzięki temu kod staje się bardziej przejrzysty i sprecyzowany. Nie należy tak robić. Jednym z powodów stosowania języków wysokiego poziomu jest fakt, że to one wykonują za nas pewne rzeczy. Umieszczanie słowa kluczowego `this` tam, gdzie nie jest ono potrzebne, może zmylić i rozzłościć osoby czytające kod, gdyż we wszystkich pozostałych analizowanych przez nich kodach słowo kluczowe `this` nie będzie używane w taki sposób. Stosowanie spójnego i prostego stylu kodowania zaoszczędza czas i pieniądze.

Słowo kluczowe this przydaje się też przy przekazywaniu bieżącego obiektu do wywołań innych metod:

```
//: initialization/PassingThis.java

class Person {
  public void eat(Apple apple) {
    Apple peeled = apple.getPeeled();
    System.out.println("Pyszne");
  }
}

class Peeler {
  static Apple peel(Apple apple) {
    // ... usuwanie skórki
    return apple; // Obrane
  }
}

class Apple {
  Apple getPeeled() { return Peeler.peel(this); }
}

public class PassingThis {
  public static void main(String[] args) {
    new Person().eat(new Apple());
  }
} /* Output:
Pyszne
*///:~
```

Klasa Apple musi wywołać metodę Peeler.peel() będącą zewnętrzną metodą realizującą operację pomocniczą dla klasy Apple, która z jakichś przyczyn została wyodrębniona poza klasę Apple (choćby dlatego, że metoda ma mieć zastosowanie również w innych klasach). Aby przekazać obiekt klasy Apple do metody zewnętrznej, trzeba skorzystać ze słowa kluczowego this.

Ćwiczenie 8. Napisz klasę z dwiema metodami. W pierwszej z nich wywołaj dwukrotnie drugą metodę: za pierwszym razem bez, za drugim razem z poprzedzającym słowem kluczowym this — tylko po to, aby sprawdzić, czy to zadziała; w praktyce nie korzysta się z drugiego sposobu (1).

Wywoływanie konstruktorów z konstruktorów

Jeśli napiszemy kilka konstruktorów dla klasy, czasami chcielibyśmy wywołać jeden konstruktor z drugiego, aby uniknąć powtarzania kodu. Można to zrobić, stosując słowo kluczowe this.

Zwykle piszemy this w sensie „ten obiekt" lub „aktualny obiekt" i dzięki temu dostajemy referencję do aktualnego obiektu. W konstruktorze słowo kluczowe this nabiera innego znaczenia — jeśli poda mu się listę argumentów, to spowoduje wywołanie konstruktora, do którego pasuje ta lista. W ten sposób zyskujemy prosty sposób wywołania innych konstruktorów:

```
//: initialization/Flower.java
// Wywoływanie konstruktorów za pośrednictwem referencji "this"
import static net.mindview.util.Print.*;

public class Flower {
  int petalCount = 0;
  String s = "wartość początkowa";
  Flower(int petals) {
    petalCount = petals;
    print("Konstruktor z argumentem int, petalCount= "
      + petalCount);
  }
  Flower(String ss) {
    print("Konstruktor z argumentem String, s = " + ss);
    s = ss;
  }
  Flower(String s, int petals) {
    this(petals);
//!    this(s); // Nie można wywołać drugiego!
    this.s = s; // Kolejne zastosowanie "this"
    print("Argumenty String i int");
  }
  Flower() {
    this("hej", 47);
    print("konstruktor domyślny (bez argumentów)");
  }
  void printPetalCount() {
//! this(11); // Nie w metodzie niebędącej konstruktorem!
    print("petalCount = " + petalCount + " s = "+ s);
  }
  public static void main(String[] args) {
    Flower x = new Flower();
    x.printPetalCount();
  }
} /* Output:
Konstruktor z argumentem int, petalCount= 47
Argumenty String i int
konstruktor domyślny (bez argumentów)
petalCount = 47 s = hej
*///:~
```

Konstruktor kwiatu `Flower(String s, int petals)` pokazuje, że można w konstruktorze klasy wywołać jeden inny konstruktor, stosując `this`, nie można jednak wywołać dwóch. Ponadto wywołanie konstruktora musi być pierwszą rzeczą, którą robimy. W innym przypadku dostaniemy od kompilatora komunikat o błędzie.

Przykład ten pokazuje jeszcze jeden sposób użycia `this`. Ponieważ nazwa argumentu `s` oraz nazwa zmiennej składowej `s` są takie same, powstaje niejednoznaczność. Można ją jednak rozwiązać poprzez napisanie `this.s`, by odwołać się do składowej. Często będziesz mógł zobaczyć taką konstrukcję w kodzie Javy, jak również w wielu miejscach tej książki.

W metodzie `printPetalCount()` widać, że kompilator nie pozwoli na wywołanie konstruktora z wnętrza jakiejkolwiek metody nie będącej konstruktorem.

Ćwiczenie 9. Napisz klasę z dwoma (przeciążonymi) konstruktorami. Za pomocą słowa `this` wywołaj drugi konstruktor w pierwszym (1).

Znaczenie słowa static

Pamiętając o słowie kluczowym `this`, można lepiej zrozumieć, co to znaczy uczynić metodę statyczną. Oznacza to, że dla tej konkretnej metody w ogóle nie istnieje `this`. Nie można wywołać metod niestatycznych z wnętrza metod statycznych[2] (chociaż w drugą stronę jest to możliwe), ale można wywołać metodę `static` na rzecz samej klasy, nie wykorzystując żadnego obiektu. Faktycznie jest to zasadniczy cel metody typu `static`. To tak, jakby stworzyć odpowiednik metody globalnej — poza tym, że metody globalne nie są w Javie dozwolone, a zamieszczenie metody statycznej wewnątrz klasy pozwala jej na dostęp do innych metod i składowych statycznych.

Niektórzy twierdzą, że metody statyczne nie są zorientowane obiektowo, gdyż mają semantykę metod globalnych; w przypadku metod statycznych nie przesyłamy komunikatu do obiektu, ponieważ nie istnieje `this`. Jest to prawdopodobnie słuszny argument i jeśli stwierdzimy, że używamy zbyt wielu metod statycznych, powinniśmy ponownie przemyśleć obraną strategię. Elementy statyczne reprezentują jednak podejście pragmatyczne i zdarzają się sytuacje, gdy są rzeczywiście potrzebne. Zatem pytanie, czy są one „właściwe dla programowania zorientowanego obiektowo?", powinno zostać pozostawione teoretykom.

Sprzątanie: finalizacja i odśmiecanie pamięci

Programiści zdają sobie sprawę z wagi inicjalizacji, ale często zapominają o znaczeniu „sprzątania". W końcu komu potrzebne jest niszczenie elementów typu `int`? Jednak w przypadku bibliotek proste „zezwolenie na pozostanie" obiektu po zakończeniu korzystania z niego nie zawsze jest bezpieczne. Oczywiście Java posiada odśmiecacz odzyskujący pamięć po obiektach, które nie są już wykorzystywane, rozważmy jednak niecodzienny przypadek. Przypuśćmy, że nasz obiekt alokuje „specjalną" pamięć bez stosowania operatora `new`. Odśmiecacz pamięci wie tylko, jak zwolnić pamięć przydzieloną z użyciem `new`, nie potrafi więc zwolnić pamięci „specjalnej" tego obiektu. Aby obsłużyć taki przypadek, Java dostarcza metodę o nazwie `finalize()`, którą można zdefiniować we własnej klasie. Oto jak *powinna* ona działać. Kiedy odśmiecacz pamięci jest gotowy do zwolnienia obszaru zajmowanego przez obiekt, najpierw wywołuje metodę `finalize()` tego obiektu, a dopiero przy kolejnym odśmieczaniu zwolni pamięć zajmowaną przez obiekt. Zatem jeżeli zdecydujemy się na użycie metody `finalize()`, będziemy mieli możliwość przeprowadzenia istotnego sprzątania *w czasie odśmiecania pamięci*.

[2] Jedyny przypadek, w którym jest to możliwe, to ten, kiedy przekażemy referencję do obiektu metodzie statycznej. Wtedy, poprzez tę referencję (która jest teraz faktycznie jak `this`), można wywoływać metody niestatyczne i sięgać do niestatycznych pól. Ale zazwyczaj, chcąc zrobić coś takiego, po prostu tworzymy zwyczajną, niestatyczną metodę.

Jest to potencjalna pułapka, ponieważ niektórzy programiści, szczególnie programiści C++, mogą początkowo źle rozumieć metodę `finalize()` — jako odpowiednik destruktora z C++, który jest metodą wywoływaną zawsze podczas niszczenia obiektu. Ważne jest, aby rozróżnić tu C++ i Javę, ponieważ w C++ *obiekty są zawsze niszczone* (w programach wolnych od błędów), podczas gdy w Javie obiekty nie zawsze podlegają odśmiecaniu, albo mówiąc inaczej:

1. *Obiekty mogą nie zostać poddane odśmiecaniu.*
2. *Odśmiecanie pamięci to nie to samo co destrukcja obiektów.*

Jeśli to zapamiętasz, unikniesz kłopotów. Oznacza to, że jeśli istnieje jakaś czynność, która musi zostać wykonana, zanim obiekt nie będzie już potrzebny, to trzeba ją wykonać samemu. W Javie nie ma destruktora ani podobnego narzędzia, dlatego trzeba stworzyć zwyczajną metodę do przeprowadzenia tego sprzątania. Przypuśćmy na przykład, że w procesie tworzenia obiektu odrysowuje się on na ekranie. Jeżeli jawnie nie wymaże się jego obrazu z ekranu, to może się zdarzyć, że nigdy nie zostanie wyczyszczony. Jeśli zamieścimy wywołanie jakiejś funkcji wymazującej wewnątrz metody `finalize()`, to wtedy obiekt zostanie wymazany, jeśli stanie się przedmiotem odśmiecania. Jeżeli jednak tak się nie stanie — wtedy jego obraz pozostanie.

Może się okazać, że pamięć obiektu nigdy nie zostanie zwolniona, ponieważ program nigdy nie zbliży się do sytuacji wyczerpania zasobów pamięci. Jeżeli program się zakończy i odśmiecacz pamięci nie zadziała, to pamięć ta zostanie zwrócona do systemu operacyjnego *w całości* przy wyjściu z programu. Jest to dobre rozwiązanie, ponieważ odśmiecanie pamięci ma pewien narzut, więc jeśli nigdy się go nie używa, to nigdy nie ponosi się tych kosztów.

Do czego służy finalize()

Można by przypuszczać, że nie powinno się wykorzystywać metody `finalize()` jako uniwersalnej metody sprzątającej. Do czego więc służy?

Trzecim punktem do zapamiętania jest:

3. *Odśmiecanie dotyczy wyłącznie pamięci.*

Zatem jedynym powodem istnienia odśmiecacza pamięci jest odzyskiwanie pamięci, której program już nie wykorzystuje. Toteż każde działanie związane z odśmiecaniem pamięci, a zwłaszcza metoda `finalize()`, musi dotyczyć tylko pamięci i jej zwalniania.

Czy znaczy to, że jeśli obiekt zawiera inne obiekty, to `finalize()` powinna jawnie zwalniać te obiekty? Otóż nie — odśmiecacz pamięci dba o zwolnienie pamięci wszystkich obiektów, niezależnie od sposobu, w jaki obiekt został stworzony. Wynika z tego, że potrzeba istnienia metody `finalize()` ogranicza się do specjalnych przypadków, w których obiekt może alokować obszar pamięci inaczej niż poprzez stworzenie obiektu. Skoro można sądzić, że wszystko w Javie jest obiektem, jak to jest możliwe?

Wynika z tego, że `finalize()` istnieje ze względu na możliwość wykonania czegoś w stylu języka C — alokacji pamięci za pomocą mechanizmu innego niż normalny dla Javy.

Może się to odbywać poprzez *metody natywne*, które są sposobem na wywołanie z Javy kodu nienapisanego w Javie (metody natywne są opisane w dodatku B drugiego wydania książki, które, w wersji elektronicznej, można znaleźć w witrynie *www.MindView.net*). Języki C i C++ są aktualnie jedynymi językami obsługiwanymi przez metody natywne, ale ponieważ mogą one wywoływać podprogramy w innych językach, toteż efektywnie można wywołać wszystko. Wewnątrz kodu obcego można wywołać funkcję z rodziny `malloc()` z języka C do alokacji pamięci i, dopóki nie wywołamy `free()`, pamięć ta nie zostanie zwolniona, powodując wyciekanie pamięci. Ponieważ `free()` jest funkcją C i C++, trzeba by ją wywołać w metodzie natywnej wewnątrz własnej metody `finalize()`.

Po przeczytaniu tego prawdopodobnie doszedłeś do wniosku, że nie będziesz zbyt często wykorzystywał `finalize()`[3]. Masz rację, nie jest to odpowiednie narzędzie do przeprowadzenia normalnych porządków. Jak zatem odbywają się owe normalne porządki?

Musisz przeprowadzić sprzątanie

Aby posprzątać po obiekcie, jego użytkownik musi wywołać metodę sprzątającą w miejscu, w którym takie czyszczenie jest pożądane. Wydaje się to łatwe, ale nieco koliduje z ideą destruktora znaną z C++. Bowiem w C++ wszystkie obiekty są niszczone — albo raczej wszystkie obiekty *powinny* być niszczone. Jeśli obiekt C++ zostanie stworzony jako lokalny (tj. na stosie — nie jest to możliwe w Javie), wtedy destrukcja następuje przy nawiasie klamrowym zamykającym zasięg, w którym obiekt został stworzony. Jeśli obiekt był stworzony z użyciem `new` (tak jak w Javie), to destruktor jest wywoływany, kiedy programista użyje operatora `delete` (w Javie nie istnieje taki). Jeśli programista C++ zapomni wywołać `delete`, destruktor nigdy nie zostanie wywołany, co spowoduje powstanie wycieku pamięci oraz to, że inne części obiektu nigdy nie zostaną poddane sprzątaniu. Tego rodzaju błędy mogą być bardzo trudne do wytropienia, a możliwość ich uniknięcia jest nieodpartym argumentem przemawiającym za porzuceniem C++ i rozpoczęciem korzystania z Javy.

W przeciwieństwie do C++ Java nie pozwala na tworzenie obiektów lokalnych — zawsze trzeba użyć `new`. Ale w Javie nie istnieje `delete`, które można by wywołać w celu zwolnienia obiektu, ponieważ odśmiecacz zwalnia pamięć za nas. Zatem, upraszczając, można powiedzieć, że dzięki odśmiecaczowi pamięci Java nie ma destruktorów. Przekonasz się jednak, podczas dalszej lektury, że obecność odśmiecacza pamięci nie zmienia potrzeby czy też użyteczności destruktorów (nie powinno się nigdy wywoływać metody `finalize()` bezpośrednio, zatem to nie jest rozwiązanie). Jeżeli chcemy przeprowadzić rodzaj sprzątania inny niż zwolnienie pamięci, trzeba *nadal* jawnie wywoływać odpowiednią metodę, która będzie odpowiednikiem destruktora C++, bez wygody jego posiadania.

Należy pamiętać, że ani odśmiecanie pamięci, ani finalizacja nie są zagwarantowane. Jeśli maszyna wirtualna Javy (JVM) nie zbliża się do wyczerpania zasobów pamięci, to — roztropnie — nie będzie tracić czasu na odzyskiwanie pamięci poprzez odśmiecanie.

[3] Joshua Bloch w podrozdziale „Unikaj metod finalize" posuwa się nawet dalej, pisząc: „Finalizatory są nieprzewidywalne, często niebezpieczne i, ogólnie rzecz biorąc, niepotrzebne.". Fragment książki *Effective Java™ Programming Language Guide* (Addison-Wesley, rok wydania 2001), strona 20.

Warunek zakończenia

Zasadniczo nie można polegać na wywołaniu metody `finalize()` — trzeba stworzyć oddzielne funkcje „sprzątające" i jawnie je wywoływać. Zatem okazuje się, że `finalize()` jest użyteczna wyłącznie przy zwalnianiu nieznanej odśmiecaczowi pamięci (np. pamięci zajętej w stylu C), czego większość programistów nigdy nie użyje. Istnieje jednak bardzo interesujące zastosowanie metody `finalize()`, które nie opiera się na jej wywoływaniu za każdym razem. Jest to weryfikacja *warunku zakończenia*[4] obiektu.

Kiedy obiekt (który już dłużej nas nie interesuje) jest gotowy do zniszczenia, powinien znaleźć się w takim stanie, w którym pamięć mu przydzielona może zostać bezpiecznie zwolniona. Na przykład, jeśli obiekt reprezentuje otwarty plik, to plik ten powinien zostać przez programistę zamknięty, zanim ulegnie procesowi odśmiecania pamięci. Jeśli części obiektu nie są właściwie wyczyszczone, to pojawia się w programie błąd, który bardzo trudno odnaleźć. Wartość `finalize()` polega na tym, że może być ona wykorzystana do odkrycia warunku zakończenia, nawet jeżeli nie zawsze jest wywoływana. Jeśli wystąpi jedna z finalizacji, aby ujawnić ewentualny błąd, odkryjemy problem i to wszystko, o co nam chodzi.

Oto prosty przykład klasy reprezentującej książkę, pokazujący, jak można to wykorzystać:

```
// : initialization/TerminationCondition.java
// Wykorzystanie finalize() do wykrywania obiektu,
// który nie został odpowiednio "sprzątnięty".

class Book {
  boolean checkedOut = false;
  Book(boolean checkOut) {
    checkedOut = checkOut;
  }
  void checkIn() {
    checkedOut = false;
  }
  protected void finalize() {
    if(checkedOut)
      System.out.println("Błąd: w obiegu");
    // Normalnie użyłbyś również tego:
    // super.finalize(); // Wywołanie wersji z klasy bazowej
  }
}

public class TerminationCondition {
  public static void main(String[] args) {
    Book novel = new Book(true);
    // Właściwe "sprzątanie":
    novel.checkIn();
    // Porzucenie referencji, przeoczenie sprzątania:
    new Book(true);
    // Wymuszenie odśmiecania pamięci i finalizacji:
    System.gc();
  }
```

[4] Termin ten został wprowadzony przez Billa Vennersa (*www.artima.com*) podczas seminarium, które razem prowadziliśmy.

```
} /* Output:
Błąd: w obiegu
*///:~
```

Warunek zakończenia polega na tym, że wszystkie obiekty klasy `Book` powinny zostać zwolnione poprzez ustawienie wartości `checkedOut` (książka „w obiegu") na `false`, zanim będą podlegać odśmiecaniu, ale w metodzie `main()` z powodu błędu programisty nie jest to robione dla jednej z książek. Bez dodania metody `finalize()`, sprawdzającej warunek śmierci, mogłoby to być błędem dosyć trudnym do odnalezienia.

Zauważ, że `System.gc()` jest używana do wymuszenia finalizacji (powinno się ją przeprowadzać podczas tworzenia programu, aby przyspieszyć testowanie). Jednak nawet gdyby nie była, to jest wysoce prawdopodobne, że zgubiony obiekt `Book` zostałby ostatecznie odkryty poprzez wielokrotne wykonywanie programu (przy założeniu, że program alokuje wystarczająco dużo pamięci, aby spowodować uruchomienie odśmiecacza pamięci).

Zasadniczo powinieneś zawsze zakładać, że wersja `finalize()` z klasy bazowej realizuje pewne ważne dla porządkowania operacje i każdorazowo przy finalizacji wywoływać tę wersję za pośrednictwem referencji `super`, jak w metodzie `Book.finalize()`. Tam wywołanie wersji z klasy bazowej zostało oznaczone jako komentarz, bo wymagałoby obsługi wyjątków, której jeszcze nie opanowałeś.

Ćwiczenie 10. Napisz klasę z metodą `finalize()`, która wypisze komunikat na wyjściu programu. W metodzie `main()` utwórz obiekt klasy. Wyjaśnij zaobserwowane działanie programu (2).

Ćwiczenie 11. Zmodyfikuj poprzednie ćwiczenie tak, aby metoda `finalize()` była wywoływana za każdym razem (4).

Ćwiczenie 12. Utwórz klasę o nazwie `Tank` (zbiornik) obsługującą operacje wypełniania (`fill()`) i opróżniania (`empty()`) wyposażoną w *warunek zakończenia*, wymagający opróżnienia obiektu przed usunięciem. Napisz metodę `finalize()`, która będzie sprawdzać stan zbiornika. W metodzie `main()`, sprawdź możliwe scenariusze używania obiektów klasy `Tank` (4).

Jak działa odśmiecacz pamięci

Jeśli korzystałeś z języka programowania, w którym alokowanie obiektów na stercie było kosztowne, to naturalnie możesz przypuszczać, że pomysł alokowania wszystkiego (poza typami podstawowymi) na stercie w przypadku Javy również jest taki. Jednak okazuje się, że odśmiecacz pamięci może mieć znaczący wpływ na *zwiększenie* prędkości tworzenia obiektów. Może to początkowo brzmieć odrobinę dziwacznie — zwalnianie pamięci wpływa na jej przydział — ale w ten sposób działają niektóre maszyny wirtualne Javy. Okazuje się, że alokowanie pamięci na stercie w Javie może być prawie tak szybkie, jak alokowanie obszaru pamięci na stosie w przypadku innych języków.

Stertę C++ można wyobrazić sobie jako plac, na którym każdy obiekt zajmuje swój własny kawałek miejsca. Później miejsce to może zostać zwolnione i ponownie wykorzystane. W niektórych JVM sterta Javy jest zupełnie inna — jest bardziej podobna do taśmy

produkcyjnej, która przesuwa się do przodu za każdym razem, gdy przydzielimy pamięć nowemu obiektowi. Dzięki temu alokacja pamięci dla obiektu jest bardzo szybka. „Wskaźnik sterty" jest zwyczajnie przesuwany do przodu, do rejonów jeszcze nietkniętych, a więc w efekcie wygląda to tak samo, jak w przypadku przydziału na stosie w C++ (oczywiście jest tu trochę dodatkowego narzutu dla obliczeń, ale to nic w porównaniu z szukaniem obszaru pamięci).

Możesz zauważyć, że sterta w Javie w rzeczywistości nie jest taśmą, a jeśli traktuje się ją w ten sposób, może to doprowadzić do częstej wymiany stron pamięci (co bardzo wpływa na wydajność), a następnie do jej wyczerpania. Sztuczka polega na tym, że do działania wkracza odśmiecacz pamięci i podczas odzyskiwania pamięci umieszcza wszystkie obiekty w ciągłym obszarze sterty, a więc faktycznie przesuwamy „wskaźnik stosu" bliżej początku taśmy i unikamy sytuacji błędu braku strony. Odśmiecacz pamięci przestawia obiekty oraz umożliwia alokację pamięci z dużą prędkością w modelu sterty z nieograniczoną pamięcią.

Aby zrozumieć, jak to działa, należy lepiej poznać sposoby pracy różnych odśmiecaczy pamięci. Prosta, ale wolna, technika to *zliczanie referencji*. Polega ona na tym, że każdy obiekt zawiera licznik odwołań, który za każdym razem, gdy następuje przypisanie obiektu do referencji, jest zwiększany o jeden. Z kolei kiedy referencja wskazująca na obiekt wychodzi poza zasięg lub jest ustawiana na `null`, licznik odwołań do tego obiektu jest zmniejszany. W ten sposób zarządzanie licznikami odwołań powoduje niewielki, ale stały narzut, który występuje przez cały czas działania programu. Odśmiecacz pamięci przegląda całą listę obiektów i, kiedy znajdzie obiekt z licznikiem o wartości zero, zwalnia odpowiadający mu obszar pamięci (tyle że w technikach zliczania referencji usuwanie obiektu realizuje się zwykle tuż po stwierdzeniu wyzerowania licznika odwołań, bez oczekiwania na wszczęcie procedury odśmiecania). Poważna wada tego rozwiązania ujawnia się w przypadku wystąpienia cyklu w odwołaniach pomiędzy obiektami — w takiej sytuacji obiekty te powinny zostać poddane odśmiecaniu mimo niezerowych liczników odwołań. Jednak lokalizacja takich cyklicznych grup wymagałaby od odśmiecacza wiele dodatkowej pracy. Zliczanie referencji jest powszechnie wykorzystywane do wyjaśnienia zasady działania jednego z rodzajów odśmiecania pamięci, ale nie wydaje się być używane w żadnej z implementacji JVM.

Szybsza odmiana odśmiecania pamięci nie bazuje na zliczaniu odwołań. Zamiast tego opiera się na założeniu, że każdy „żywy" obiekt musi prowadzić z powrotem do referencji, która znajduje się albo na stosie, albo w obszarze zmiennych statycznych. Łańcuszek może prowadzić przez kilka warstw obiektów. Tak więc, jeśli wystartujemy ze stosu i obszaru pamięci statycznej, przechodząc przez wszystkie odwołania, to odnajdziemy wszystkie żywe obiekty. Musimy śledzić obiekty wskazywane przez wszystkie referencje, które napotkamy, a następnie wszystkie referencje występujące *w tych* obiektach. Jeszcze później musimy śledzić obiekty, na które one wskazują itd., dopóki nie przejdziemy przez całą sieć powstałą z referencji znajdujących się na stosie lub w pamięci statycznej. Każdy obiekt, który odwiedzamy, musi być wciąż aktywny. Nie ma problemu z grupami cyklicznymi — ich po prostu nie napotkamy i dlatego zostaną automatycznie uprzątnięte.

W przedstawianym tutaj rozwiązaniu wirtualna maszyna Javy używa *adaptacyjnego* modelu odśmiecania pamięci i to, co robi ze znalezionymi obiektami aktywnymi, zależy od zastosowanego wariantu. Jedną z możliwości jest *zatrzymaj i kopiuj* (ang. *stop-and-copy*).

Zatem — z powodu, który za chwilę okaże się oczywisty — program jest najpierw zatrzymywany (nic jest to odśmiecanie w tle), a następnie każdy znaleziony obiekt aktywny jest kopiowany z jednej sterty na inną, a wszystkie śmieci (obiekty nieaktywne) są pozostawiane na poprzedniej stercie. Dodatkowo podczas kopiowania obiektów na nową stertę są one umieszczane na niej w ten sposób, by ją gęsto wypełniać (dzięki temu możliwe jest późniejsze rozwijanie obszaru obiektów od końca sterty, tak jak to opisano wcześniej).

Oczywiście gdy obiekt jest przesuwany z jednego miejsca w drugie, to wszystkie odwołania wskazujące na obiekt (tj. takie *referencje*) muszą ulec zmianie. Referencje do obiektu znajdujące się na stercie lub w obszarze pamięci statycznej mogą zostać zmienione natychmiast, ale mogą istnieć inne referencje, wskazujące na ten sam obiekt, które zostaną napotkane później podczas „przechodzenia". Są one modyfikowane w momencie znalezienia (można sobie wyobrazić tablicę odwzorowującą stare adresy na nowe).

Istnieją dwie przyczyny, które powodują, że te odśmiecacze, zwane „odśmiecaczami kopiującymi", są mało skuteczne. Pierwsza to pomysł posiadania dwóch stert (zajmujących dwa razy więcej pamięci niż potrzeba) i kopiowanie całej pamięci w tę i z powrotem pomiędzy nimi. Niektóre JVM radzą sobie z tym poprzez alokowanie sterty po kawałku w miarę potrzeb i kopiowanie z jednego kawałka do drugiego.

Druga sprawa to samo kopiowanie. Gdy działanie naszego programu ustabilizuje się, to może on nie generować żadnych śmieci lub generować ich niewiele. Mimo to odśmiecacz kopiujący nadal będzie kopiował całą pamięć z jednego miejsca w drugie, co oczywiście jest nieefektywne. Aby temu zapobiec, niektóre maszyny wirtualne rozpoznają, że liczba nieużywanych obiektów nie wzrasta, i przełączają się na inny schemat działania (to jest właśnie część „adaptacyjna"). Ten drugi schemat pracy jest zwany *zaznacz i zamieć* (ang. *mark-and-sweep*) i to właśnie on był stosowany we wcześniejszych wersjach JVM firmy Sun. Do powszechnego użytku „zaznacz i zamieć" jest dosyć wolne, ale jeśli wiadomo, że generujemy niewiele śmieci lub w też ogóle, to okazuje się być szybkie.

„Zaznacz i zamieć" stosuje tę samą logikę rozpoczęcia przeglądania od stosu i statycznej pamięci oraz śledzenia wszystkich referencji, aby odnaleźć używane obiekty. Jednak za każdym razem, gdy odnajdzie obiekt, jest on znakowany poprzez ustawienie mu znacznika, przy czym jeszcze nie jest niszczony. Dopiero gdy proces znakowania zostanie zakończony, następuje zamiatanie — wtedy martwe obiekty są zwalniane. Jednak nic występuje tu kopiowanie, więc jeśli odśmiecacz zdecyduje się na zagęszczenie sterty, może uczynić to tylko poprzez żonglowanie obiektami.

„Zatrzymaj i kopiuj" opiera się na pomyśle *nie*wykonywania odśmiecania w tle; zamiast tego program jest zatrzymywany na czas pracy odśmiecacza. W literaturze firmy Sun można znaleźć wiele odwołań do odśmiecania pamięci jako procesu działającego w tle o niskim priorytecie, ale okazuje się, że odśmiecanie nie było implementowane w ten sposób, przynajmniej we wcześniejszych wersjach maszyn wirtualnych Suna. Zamiast tego odśmiecacz pamięci Suna był uruchamiany, gdy zaczynało brakować pamięci. Na dodatek „zaznacz i zamieć" wymaga zatrzymania programu.

Jak wspominałem wcześniej, w JVM tutaj opisywanej pamięć jest przydzielana w dużych blokach. Jeśli zaalokuje się duży obiekt, to dostaje on swój własny blok. Ścisłe „zatrzymaj i kopiuj" wymaga kopiowania każdego używanego obiektu ze sterty źródłowej do nowej sterty przed możliwością zwolnienia, co przekłada się na dużą ilość potrzebnej

pamięci. W przypadku stosowania bloków można wykorzystać martwe bloki do kopiowania obiektów podczas odśmiecania. Każdy blok posiada *licznik generacji* (inaczej *licznik pokolenia*) do kontroli tego, czy jest „żywy". W normalnym przypadku tylko bloki stworzone od czasu ostatniego przebiegu głównego odśmiecania są przeglądane i zagęszczane; wszystkie inne bloki zwiększają swoje liczniki generacji, jeśli były do nich odwołania z innych miejsc. Dotyczy to normalnego przypadku wielu krótko używanych, tymczasowych obiektów. Okresowo przeprowadza się pełne sprzątanie — duże obiekty nadal nie są kopiowane (tylko zwiększa się ich licznik generacji), a bloki zawierające małe obiekty są kopiowane do ciągłego obszaru pamięci. JVM monitoruje efektywność odśmiecania i, jeśli staje się to stratą czasu, ponieważ wszystkie obiekty są długo wykorzystywane, przełącza się w tryb „zaznacz i zamieć". Podobnie maszyna wirtualna kontroluje, jak efektywne jest działanie „zaznacz i zamieć", oraz — gdy sterta zaczyna być zbyt pofragmentowana — przełącza się z powrotem na pracę w trybie „zatrzymaj i kopiuj". To właśnie jest część „przystosowawcza", zatem skończyliśmy na schemacie: „adaptacyjny, generacyjny „zatrzymaj i kopiuj" oraz „zaznacz i zamieć"".

Istnieje wiele dodatkowych możliwych do implementacji „przyspieszaczy" JVM. Szczególnie ważny angażuje działanie mechanizmu ładowania klas (ang. *class loader*) oraz kompilatora Just-In-Time (JIT). Gdy klasa musi zostać załadowana (przeważnie za pierwszym razem, gdy chcemy stworzyć obiekt tej klasy), lokalizowany jest odpowiadający jej plik *.class*, a kod bajtowy klasy jest przenoszony do pamięci. W tym momencie jednym z rozwiązań jest użycie JIT dla całego kodu, ale ma ono dwie wady: zabiera trochę więcej czasu, co w stosunku do całego czasu działania programu może być znaczące, oraz zwiększa rozmiar pliku wynikowego (kod bajtowy jest znacznie bardziej zwarty niż rozrośnięty kod uzyskiwany po kompilacji z użyciem JIT), co może powodować bardziej intensywną wymianę stron i tym samym zdecydowanie spowolnić działanie programu. Alternatywnym podejściem jest *opóźniona ewaluacja*, która oznacza, że kod nie jest kompilowany przez JIT, dopóki nie jest to konieczne. Zatem kod, który może nigdy nie być wykonany, może również nigdy nie być kompilowany za pomocą JIT. Technologie Java HotSpot stosowane w najnowszych wersjach JDK stosują podobne rozwiązanie, w coraz większym stopniu optymalizując fragment kodu za każdym razem, gdy jest on wykonywany; a zatem im częściej kod jest wykonywany, tym staje się szybszy.

Inicjalizacja składowych

Java gwarantuje, że zmienne zostaną właściwie zainicjowane przed próbą ich użycia. W przypadku zmiennych definiowanych lokalnie dla metod ta gwarancja objawia się błędem kompilacji. Zatem jeśli napiszemy:

```
void f() {
    int i;
    i++;    // Błąd — zmienna i nie jest zainicjalizowana
  }
```

dostaniemy komunikat o błędzie, mówiący że i może nie być zainicjowana. Oczywiście kompilator mógłby nadać zmiennej wartość domyślną, ale jest bardziej prawdopodobne, że jest to błąd programisty, i taka wartość domyślna by go ukryła. Zmuszenie programisty do dostarczenia wartości początkowej zwiększa prawdopodobieństwo wykrycia błędu.

Jeśli zmienna typu podstawowego jest składową klasy, to sytuacja jest trochę inna. W rozdziale „Wszystko jest obiektem" padło stwierdzenie, że wszystkie składowe klasy typów podstawowych objęte są gwarantowaną inicjalizacją. Wartości wykorzystywane w takiej inicjalizacji można zobaczyć w przykładzie:

```
//: initialization/InitialValues.java
// Pokazuje domyślne wartości początkowe składowych typów podstawowych.
import static net.mindview.util.Print.*;

public class InitialValues {
  boolean t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  InitialValues reference;
  void printInitialValues() {
    print("Typ danych     Wartość początkowa");
    print("boolean        " + t);
    print("char           [" + c + "]");
    print("byte           " + b);
    print("short          " + s);
    print("int            " + i);
    print("long           " + l);
    print("float          " + f);
    print("double         " + d);
    print("referencja     " + reference);
  }
  public static void main(String[] args) {
    InitialValues iv = new InitialValues();
    iv.printInitialValues();
    /* Można by też napisać:
    new InitialValues().printInitialValues();
    */
  }
} /* Output:
Typ danyc    Wartość początkowa
boolean      false
char         [ ]
byte         0
short        0
int          0
long         0
float        0.0
double       0.0
referencja   null
*///:~
```

Mimo że nie określono wartości dla zmiennych, zostaną one automatycznie zainicjowane (wartością zmiennej typu char jest zero, co powoduje wypisanie znaku odstępu), a zatem przynajmniej nie ma niebezpieczeństwa pracy z niezainicjowanymi zmiennymi.

Jeżeli wewnątrz klasy zdefiniujemy referencję do obiektu bez przeprowadzenia jej inicjalizacji, to referencja ta uzyska specjalną wartość `null`.

Określanie sposobu inicjalizacji

Co się stanie, jeżeli zechcemy nadać zmiennej wartość początkową? Bezpośrednim sposobem zrobienia tego jest po prostu przypisanie wartości w miejscu definicji zmiennej wewnątrz klasy (zauważ, że nie można tego zrobić w C++, chociaż nowicjusze zawsze próbują). Poniższe definicje pól klasy `InitialValues` zostały zmienione tak, aby zapewnić wartości początkowe:

```
//: initialization/InitialValues2.java
// Jawne podawanie wartości początkowych.

public class InitialValues2 {
  boolean bool = true;
  char ch = 'x';
  byte b = 47;
  short s = 0xff;
  int i = 999;
  long lng = 1;
  float f = 3.14f;
  double d = 3.14159;
} ///:~
```

Można w ten sam sposób zainicjować także składowe będące obiektami (a nie wartościami typów podstawowych). Jeśli `Depth` jest jakąś klasą, to można dołożyć składową tej klasy i zainicjować ją następująco:

```
//: initialization/Measurement.java
class Depth {}

public class Measurement {
  Depth d = new Depth();
  // ...
} ///:~
```

Jeżeli nie nadamy referencji `d` wartości początkowej i spróbujemy jej mimo wszystko użyć, to otrzymamy komunikat o błędzie czasu wykonania zwany *wyjątkiem* (wyjątkom poświęcony jest rozdział „Obsługa błędów za pomocą wyjątków").

Można nawet wywołać metodę, aby uzyskać wartość początkową:

```
//: initialization/MethodInit.java
public class MethodInit {
  int i = f();
  int f() { return 11; }
} ///:~
```

Metoda taka może oczywiście posiadać argumenty, ale nie mogą one być jeszcze niezainicjalizowanymi składowymi klasy. Tak więc możemy napisać:

```
//: initialization/MethodInit2.java
public class MethodInit2 {
  int i = f();
  int j = g(i);
  int f() { return 11; }
  int g(int n) { return n * 10; }
} ///:~
```

ale nie możemy:

```
//: initialization/MethodInit3.java
public class MethodInit3 {
  //! int j = g(i); // Niedozwolona referencja wyprzedzająca
  int i = f();
  int f() { return 11; }
  int g(int n) { return n * 10; }
} ///:~
```

W tym przypadku kompilator *zgłosi* błąd wyprzedzonego odwołania do zmiennej, gdyż odwołania takie można wykonywać jedynie w kolejności inicjalizacji, a nie tak, jak program jest kompilowany.

Ten sposób inicjalizacji jest łatwy i prosty. Ogranicza się do tego, że *każdy* obiekt typu `InitialValues` uzyska takie same wartości początkowe. Czasami dokładnie tego potrzebujemy, czasami jednak przydałaby się większa elastyczność.

Inicjalizacja w konstruktorze

Do przeprowadzenia inicjalizacji można wykorzystać konstruktor. Pozwala to na sporą elastyczność programowania, ponieważ w celu określenia wartości początkowej można wywołać metodę czy też przeprowadzić inne działania w czasie wykonania. Trzeba przy tym pamiętać, że nie wyklucza to automatycznej inicjalizacji, która występuje przed wejściem do konstruktora. Zatem jeśli na przykład napiszemy:

```
//: initialization/Counter.java
public class Counter {
  int i;
  Counter() { i = 7; }
  // ...
} ///:~
```

to zmienna `i` zostanie najpierw zainicjowana na 0, a dopiero potem na 7. Dzieje się tak w przypadku wszystkich zmiennych typów podstawowych oraz referencji do obiektów, włączając te, które są inicjowane jawnie w miejscu definicji. Z tego względu kompilator nie zmusza nas do inicjalizacji elementów w konstruktorze w jakimś określonym miejscu albo przed wykorzystaniem — inicjalizacja była już przeprowadzona.

Kolejność inicjalizacji

Kolejność inicjalizacji wewnątrz klasy jest wyznaczona przez kolejność definiowania zmiennych w danej klasie. Definicja zmiennych może być rozsiana po całym ciele klasy oraz między definicjami metod, ale zawsze zmienne są inicjowane, zanim może zostać wywołana jakakolwiek metoda — nawet konstruktor. Na przykład:

```
//: initialization/OrderOfInitialization.java
// Demonstruje kolejność inicjalizacji.
import static net.mindview.util.Print.*;
```

```
// Kiedy do utworzenia obiektu Window wywoływany jest konstruktor,
// program wypisuje komunikat na wyjściu:
class Window {
  Window(int marker) { print("Window(" + marker + ")"); }
}

class House {
  Window w1 = new Window(1); // Przed konstruktorem
  House() {
    // Pokażmy, że jesteśmy w konstruktorze:
    print("House()");
    w3 = new Window(33); // Ponowna inicjalizacja w3
  }
  Window w2 = new Window(2); // Za konstruktorem
  void f() { print("f()"); }
  Window w3 = new Window(3); // Na końcu
}

public class OrderOfInitialization {
  public static void main(String[] args) {
    House h = new House();
    h.f(); // Pokazuje, że konstrukcja się odbyła
  }
} /* Output:
Window(1)
Window(2)
Window(3)
House()
Window(33)
f()
*///:~
```

Definicje obiektów `Window` w klasie `House` zostały specjalnie rozrzucone, aby udowodnić, że będą inicjalizowane przed wejściem do konstruktora i czymkolwiek innym. Dodatkowo zmienna `w3` jest ponownie inicjalizowana wewnątrz konstruktora.

Na podstawie wygenerowanych wyników można określić, że referencja `w3` została zainicjowana dwukrotnie, pierwszy raz przed i drugi raz podczas wywołania konstruktora (pierwszy obiekt nie jest używany, a więc może być poddany odśmiecaniu pamięci). Może się to początkowo wydawać niewydajne, ale gwarantuje poprawną inicjalizację — co by się stało, jeśli byłby zdefiniowany dodatkowy, przeciążony konstruktor i nie inicjował `w3` oraz nie funkcjonowałaby inicjalizacja „domyślna" podczas definicji?

Inicjalizacja zmiennych statycznych

Jak wiadomo, istnieje tylko pojedynczy fragment pamięci dla składowych statycznych, niezależnie od tego, jak wiele obiektów zostanie stworzonych. Nie można oznaczać słowem `static` zmiennych lokalnych, zostają więc jedynie składowe klas. Jeśli składowa jest statyczną wartością typu podstawowego i nie zostanie zainicjalizowana, to uzyska domyślną wartość początkową właściwą dla jej typu. Jeżeli jest referencją obiektu, zostanie domyślnie zainicjalizowana referencją pustą (`null`).

Chcąc zamieścić inicjalizację w miejscu definicji, postępujemy tak samo, jak w przypadku danych niestatycznych.

Na pytanie, *kiedy* następuje inicjalizacja danych statycznych, odpowiada poniższy przykład:

```
//: initialization/StaticInitialization.java
// Określanie wartości początkowych w definicji klasy.
import static net.mindview.util.Print.*;

class Bowl {
  Bowl(int marker) {
    print("Bowl(" + marker + ")");
  }
  void f1(int marker) {
    print("f1(" + marker + ")");
  }
}

class Table {
  static Bowl bowl1 = new Bowl(1);
  Table() {
    print("Table()");
    bowl2.f1(1);
  }
  void f2(int marker) {
    print("f2(" + marker + ")");
  }
  static Bowl bowl2 = new Bowl(2);
}

class Cupboard {
  Bowl bowl3 = new Bowl(3);
  static Bowl bowl4 = new Bowl(4);
  Cupboard() {
    print("Cupboard()");
    bowl4.f1(2);
  }
  void f3(int marker) {
    print("f3(" + marker + ")");
  }
  static Bowl bowl5 = new Bowl(5);
}

public class StaticInitialization {
  public static void main(String[] args) {
    print("Tworzenie nowego obiektu Cupboard() wewnątrz main");
    new Cupboard();
    print("Tworzenie nowego obiektu Cupboard() wewnątrz main");
    new Cupboard();
    table.f2(1);
    cupboard.f3(1);
  }
  static Table table = new Table();
  static Cupboard cupboard = new Cupboard();
} /* Output:
Bowl(1)
Bowl(2)
Table()
f1(1)
Bowl(4)
Bowl(5)
```

```
Bowl(3)
Cupboard()
f1(2)
Tworzenie nowego obiektu Cupboard() wewnątrz main
Bowl(3)
Cupboard()
f1(2)
Tworzenie nowego obiektu Cupboard() wewnątrz main
Bowl(3)
Cupboard()
f1(2)
f2(1)
f3(1)
*///:~
```

Klasa `Bowl` pozwala na przegląd procesu tworzenia klasy. W klasach `Table` i `Cupboard` tworzone są składowe statyczne typu `Bowl` — porozrzucane wewnątrz definicji klas. Zauważ, że w klasie `Cupboard` definiowana jest najpierw niestatyczna składowa `Bowl bowl3`, a dopiero później wersja statyczna.

Wyniki działania programu pokazują, że inicjalizacja statyczna występuje tylko wtedy, gdy jest potrzebna. Jeśli nie stworzy się obiektu `Table` i nigdy nie odwoła do `Table.bowl1` ani `Table.bowl2`, to statyczne `Bowl bowl1` i `bowl2` nigdy nie zostaną stworzone. Jednak są one inicjalizowane wyłącznie wtedy, gdy tworzony jest *pierwszy* obiekt `Table` (albo gdy nastąpi pierwsze sięgnięcie do składowej statycznej). Po tym obiekty statyczne nie są nigdy inicjalizowane.

Kolejność inicjalizacji jest następująca: najpierw zmienne `static`, jeżeli jeszcze nie zostały zainicjalizowane przez poprzednie stworzenie obiektu, a następnie obiekty niestatyczne. Dowody tego widać w wyniku działania programu z przykładu. Aby możliwe było uruchomienie statycznej metody `main()`, konieczne jest załadowanie klasy `StaticInitialization` i zainicjalizowanie jej statycznych pól `table` i `cupboard`, co z kolei wymusza załadowanie obu tych klas; ponieważ obie zawierają statyczne obiekty klasy `Bowl`, konieczne jest również załadowanie klasy `Bowl`. Wszystkie klasy składające się na ten akurat program są zatem ładowane jeszcze przed uruchomieniem metody `main()`. Nie jest to typowe zachowanie programu, bo w typowych programach rzadko konsoliduje się ze sobą wszystkie klasy programu za pośrednictwem referencji ze słowem `static`.

Przydatne będzie podsumowanie procesu tworzenia obiektów. Rozważmy więc klasę o nazwie `Dog`:

1. Gdy po raz pierwszy tworzony jest obiekt typu `Dog` *albo* gdy po raz pierwszy następuje sięgniecie do metody lub zmiennej statycznej zamieszczonej w tej klasie, interpreter Javy musi zlokalizować plik *Dog.class*, co robi poprzez przeszukiwanie ścieżki klas (ang. *classpath*).
2. Po wczytaniu pliku *Dog.class* (stworzeniu obiektu typu `Class`, o czym dowiesz się później) zostają przeprowadzone wszystkie inicjalizacje statyczne. A więc inicjalizacja statyczna występuje tylko raz, gdy obiekt klasy `Class` jest ładowany po raz pierwszy.
3. Gdy tworzymy nowy obiekt poprzez `new Dog()`, proces konstrukcji obiektu `Dog` przystępuje do alokacji odpowiedniej ilość pamięci dla obiektu na stercie.

4. Przydzielony obszar jest wymazywany z automatycznym ustawieniem wszystkich typów podstawowych obiektu Dog na ich wartości domyślne (zero w przypadku liczb i jego odpowiednik dla typu boolean oraz char), a referencji na null.
5. Następuje inicjalizacja wszystkich pól określona jawnie w miejscu ich definicji.
6. Wykonywane są ciała konstruktorów. Jak zobaczysz w rozdziale 7., może to powodować dosyć sporo działań, szczególnie gdy dochodzi jeszcze dziedziczenie.

Jawna inicjalizacja statyczna

Java pozwala na zebranie pozostałych inicjalizacji statycznych wewnątrz specjalnej „klauzuli konstrukcyjnej static" (czasami nazywanej *blokiem statycznym*) umieszczonej w klasie. Wygląda to tak:

```
//: initialization/Spoon.java
public class Spoon {
  static int i;
  static {
    i = 47;
  }
} ///:~
```

Wygląda to jak metoda, ale posiada jedynie słowo kluczowe static, po którym występuje ciało metody. Kod ten, podobnie jak inne inicjalizacje statyczne, jest wykonywany tylko raz — gdy tworzymy obiekt danej klasy lub gdy po raz pierwszy sięgamy do jednej z jej składowych statycznych (nawet jeśli nigdy nie stworzymy obiektu tej klasy). Spójrzmy na przykład:

```
//: initialization/ExplicitStatic.java
// Jawna inicjalizacja składowej statycznej w bloku statycznym.
import static net.mindview.util.Print.*;

class Cup {
  Cup(int marker) {
    print("Cup(" + marker + ")");
  }
  void f(int marker) {
    print("f(" + marker + ")");
  }
}

class Cups {
  static Cup cup1;
  static Cup cup2;
  static {
    cup1 = new Cup(1);
    cup2 = new Cup(2);
  }
  Cups() {
    print("Cups()");
  }
}

public class ExplicitStatic {
  public static void main(String[] args) {
    print("Wewnątrz main()");
```

```
    Cups.cup1.f(99);  // (1)
  }
  // static Cups cups1 = new Cups();  // (2)
  // static Cups cups2 = new Cups();  // (2)
} /* Output:
Wewnątrz main()
Cup(1)
Cup(2)
f(99)
*///:~
```

Statyczne inicjalizatory klasy Cup wykonują się, gdy nastąpi odwołanie do statycznego obiektu cup1 w wierszu oznaczonym (1) lub jeżeli wiersz ten zostanie umieszczony w komentarzu, a wiersze oznaczone przez (2) odkomentowane. Jeśli oba wiersze są wykomentowane, to statyczna inicjalizacja klasy Cup nigdy nie nastąpi — widać to na wyjściu programu. Podobnie nie ma znaczenia, czy jeden czy oba wiersze oznaczone (2) nie są wykomentowane — inicjalizacja statyczna wystąpi tylko raz.

Ćwiczenie 13. Sprawdź twierdzenia padające w ostatnich akapitach (1).

Ćwiczenie 14. Napisz klasę ze statycznym polem typu String inicjalizowanym w miejscu definicji oraz jeszcze jednym takim polem — inicjalizowanym w bloku statycznym. Dodaj do klasy metodę statyczną, która wypisze wartości obu pól i dowiedzie, że oba zostały zainicjalizowane przed użyciem (1).

Inicjalizacja egzemplarza

Java przewiduje podobną składnię, zwaną *inicjalizacją egzemplarza*, w celu inicjalizacji niestatycznych zmiennych dla każdego obiektu. Oto przykład:

```
//: initialization/Mugs.java
// "Inicjalizacja egzemplarza"
import static net.mindview.util.Print.*;

class Mug {
  Mug(int marker) {
    print("Mug(" + marker + ")");
  }
  void f(int marker) {
    print("f(" + marker + ")");
  }
}

public class Mugs {
  Mug mug1;
  Mug mug2;
  {
    mug1 = new Mug(1);
    mug2 = new Mug(2);
    print("Zainicjalizowano mug1 i mug2");
  }
  Mugs() {
    print("Mugs()");
  }
  Mugs(int i) {
```

```
    print("Mugs(int)");
  }
  public static void main(String[] args) {
    print("Wewnątrz main()");
    new Mugs();
    print("new Mugs() - zakończone");
    new Mugs(1);
    print("new Mugs(1) - zakończone");
  }
} /* Output:
Wewnątrz main()
Mug(1)
Mug(2)
Zainicjalizowano mug1 i mug2
Mugs()
new Mugs() - zakończone
Mug(1)
Mug(2)
Zainicjalizowano mug1 i mug2
Mugs(int)
new Mugs(1) - zakończone
*///:~
```

Widać, że klauzula inicjalizacji egzemplarza:

```
{
  mug1 = new Mug(1);
  mug2 = new Mug(2);
  System.out.println("Zainicjalizowano mug1 i mug2");
}
```

wygląda dokładnie tak, jak klauzula inicjalizacji statycznej poza brakującym słowem kluczowym `static`. Zapis taki jest konieczny, aby zapewnić *inicjalizację anonimowych klas wewnętrznych* (zobacz rozdział „Klasy wewnętrzne"), ale daje również gwarancję, że pewne operacje zostaną zrealizowane niezależnie od wywołania któregoś z jawnych konstruktorów. Na wyjściu programu widać, że blok inicjalizacji egzemplarza jest każdorazowo wykonywany jeszcze przed dowolnym z konstruktorów.

Ćwiczenie 15. Utwórz klasę ze składową (niestatyczną) typu `String` inicjalizowaną w bloku inicjalizacji egzemplarza klasy (1).

Inicjalizacja tablic

Tablica jest po prostu sekwencją obiektów albo zmiennych typu podstawowego, wszystkich tego samego typu, zebranych pod wspólną nazwą. Tablice definiuje się i wykorzystuje za pomocą *operatora indeksowania* `[]` — nawiasów kwadratowych. Aby zdefiniować referencję tablicy, należy przy nazwie typu umieścić puste nawiasy kwadratowe:

```
int[] a1;
```

Można również zamieścić nawiasy kwadratowe po identyfikatorze, co ma dokładnie takie samo znaczenie:

```
int a1[];
```

Jest to zgodne z oczekiwaniami programistów C i C++. Pierwszy styl prawdopodobnie jest jednak bardziej sensowny składniowo, gdyż określa typ jako „tablicę elementów int". Ten styl będę więc wykorzystywał.

Kompilator nie pozwala określić, jak duża ma być tablica. Wracamy więc do kwestii „referencji". Wszystko, co do tej pory utworzyliśmy, to referencja do tablicy — dla samej tablicy nie została jeszcze zaalokowana żadna pamięć. Aby stworzyć obszar pamięci dla tablicy, trzeba napisać wyrażenie inicjalizujące. W przypadku tablic inicjalizacja może wystąpić gdzickolwiek w kodzie, ale można również użyć specjalnego rodzaju wyrażenia inicjalizującego, występującego w miejscu deklaracji tablicy. Owo wyrażenie to lista elementów umieszczona w nawiasach klamrowych. Przydziałem pamięci (równoważność użycia new) w tym przypadku zajmie się kompilator, na przykład:

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

Zatem dlaczego mielibyśmy kiedykolwiek definiować referencję do tablicy bez samej tablicy?

```
int[] a2;
```

Cóż, możliwe jest przypisanie jednej tablicy do innej, a więc można zapisać:

```
a2 = a1;
```

Tak naprawdę dochodzi wtedy do skopiowania referencji, jak to widać tutaj:

```
//: initialization/ArraysOfPrimitives.java
import static net.mindview.util.Print.*;

public class ArraysOfPrimitives {
  public static void main(String[] args) {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.length; i++)
      a2[i] = a2[i] + 1;
    for(int i = 0; i < a1.length; i++)
      print("a1[" + i + "] = " + a1[i]);
  }
} /* Output:
a1[0] = 2
a1[1] = 3
a1[2] = 4
a1[3] = 5
a1[4] = 6
*///:~
```

Widać, że tablica a1 jest inicjalizowana, natomiast a2 nie; wartość dla a2 jest przypisywana później — w tym przypadku jest nią referencja do innej tablicy. Ponieważ a2 i a1 odnoszą się odtąd do tej samej tablicy, zmiany wartości elementów realizowane za pośrednictwem a2 są widoczne za pośrednictwem referencji a1.

Tablice posiadają wewnętrzną składową (niezależnie od tego, czy są tablicami obiektów czy typów podstawowych), którą można odczytać — ale nie zmienić — mówiącą, ile elementów znajduje się w tablicy. Składową tą jest length. Ponieważ tablice w Javie, podobnie jak w C i C++, indeksuje się od zera, największym indeksem elementu może

być `length - 1`. Jeśli wyjdziemy poza zakres tablicy, języki C i C++ spokojnie to zaakceptują i pozwolą na dostęp do dowolnego miejsca pamięci, co jest przyczyną wielu błędów. Java natomiast chroni nas przed takimi sytuacjami poprzez zgłoszenie błędu w czasie wykonania (*wyjątku*), jeśli wyjdziemy poza zakres tablicy[5].

Co zrobić, jeżeli podczas pisania programu nie wiemy, ilu elementów będziemy potrzebować w tablicy? Wystarczy użyć operatora `new` do stworzenia elementów w tablicy. Tutaj `new` działa, mimo że tworzy tablicę elementów typu podstawowego (`new` nie potrafi stworzyć zmiennych typu podstawowego w postaci innej niż tablica):

```
//: initialization/ArrayNew.java
// Tworzenie tablic za pomocą operatora new.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayNew {
  public static void main(String[] args) {
    int[] a;
    Random rand = new Random(47);
    a = new int[rand.nextInt(20)];
    print("rozmiar a = " + a.length);
    print(Arrays.toString(a));
  }
} /* Output:
rozmiar a = 18
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
*///:~
```

Rozmiar tablicy jest ustalany losowo przy użyciu metody `Random.nextInt()`, generującej wartość z zakresu od zera do wartości podanego argumentu. Losowość doboru rozmiaru eliminuje wątpliwości co do tworzenia tablicy w czasie wykonania. Dodatkowo zobaczymy na podstawie wyjścia programu, że elementy tablicy typu podstawowego są automatycznie inicjalizowane wartością „pustą" (dla liczb i typu `char` jest to zero, a dla typu `boolean` wartość `false`).

Metoda `Arrays.toString()`, wchodząca w skład biblioteki `java.util`, zwraca tekstową reprezentację zawartości tablicy jednowymiarowej, nadającą się do wypisania na wyjściu programu.

Oczywiście tablica może być również zdefiniowana i zainicjalizowana w jednej instrukcji:

```
int[] a = new int[rand.nextInt(20)];
```

To doskonałe rozwiązanie wszędzie tam, gdzie tylko da się je wykorzystać.

Tworząc tablicę wartości typu innego niż podstawowy, tworzymy tablicę referencji. Rozważmy więc typ opakowujący `Integer`, który jest klasą, a nie typem podstawowym:

[5] Oczywiście sprawdzanie każdego dostępu do tablicy zabiera trochę czasu, ale nie ma sposobu jego wyłączenia, co oznacza, że dostęp do tablic może być źródłem nieefektywności programu, jeśli występuje w jakimś krytycznym miejscu. Z powodów bezpieczeństwa internetowego i wydajności programistów projektanci Javy doszli do wniosku, że jest to wymiana opłacalna. A jeśli ktoś chciałby napisać kod mający w jego mniemaniu zwiększyć efektywność odwołań do tablic, niech sobie daruje — specjalne optymalizacje czasu kompilacji (i czasu wykonania) sprawiają, że to strata czasu.

```
// : initialization/ArrayClassObj.java
// Tworzenie tablicy obiektów klas.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayClassObj {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Integer[] a = new Integer[rand.nextInt(20)];
    print("rozmiar a = " + a.length);
    for(int i = 0; i < a.length; i++)
      a[i] = rand.nextInt(500); // Automatyczne "pakowanie" w obiekt
    print(Arrays.toString(a));
  }
} /* Output: (Sample)
rozmiar a = 18
[55, 193, 361, 461, 429, 368, 200, 22, 207, 288, 128, 51, 89, 309, 278, 498, 361, 20]
*///:~
```

Tu, nawet po wywoływaniu new, w celu utworzenia tablicy:

```
Integer[] a = new Integer[rand.nextInt(20)];
```

otrzymujemy jedynie tablicę referencji i inicjalizacja nie jest kompletna, dopóki same referencje nie zostaną zainicjalizowane przez stworzenie nowych obiektów Integer (tu akurat z użyciem mechanizmu automatycznego pakowania wartości podstawowych w obiekty):

```
a[i] = rand.nextInt(500);
```

Jeżeli zapomnimy o utworzeniu obiektu, to próba odwołania się do pustej referencji z tablicy spowoduje zgłoszenie wyjątku w czasie wykonania programu.

Możliwa jest również inicjalizacja tablicy obiektów z użyciem listy ujętej w nawiasy klamrowe. Istnieją dwie dopuszczalne formy:

```
// : initialization/ArrayInit.java
// Inicjalizacja elementów tablic.
import java.util.*;

public class ArrayInit {
  public static void main(String[] args) {
    Integer[] a = {
      new Integer(1),
      new Integer(2),
      3, // Automatyczne pakowanie w obiekty
    };
    Integer[] b = new Integer[]{
      new Integer(1),
      new Integer(2),
      3, // Automatyczne pakowanie w obiekty
    };
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
  }
} /* Output:
[1, 2, 3]
[1, 2, 3]
*///:~
```

W obu przypadkach przecinek za ostatnim elementem listy wartości inicjalizujących jest opcjonalny (ułatwia to automatyczne generowanie takich list).

Pierwsza postać jest dość użyteczna, ale ograniczona, ponieważ może zostać użyta jedynie w miejscu definicji tablicy. Postać drugą można stosować wszędzie, nawet w obrębie wywołania metody. Można w ten sposób utworzyć listę obiektów klasy `String`, przekazywanych do metody `main()`; pozwala to na wymuszanie własnych argumentów wywołania owej metody `main()`:

```
//: initialization/DynamicArray.java
// Inicjalizacja tablicy.

public class DynamicArray {
  public static void main(String[] args) {
    Other.main(new String[]{ "fiddle", "de", "dum" });
  }
}

class Other {
  public static void main(String[] args) {
    for(String s : args)
      System.out.print(s + " ");
  }
} /* Output:
fiddle de dum
*///:~
```

Tablica występująca w roli argumentu `Other.main()` jest tworzona w miejscu wywołania metody.

Ćwiczenie 16. Utwórz tablicę obiektów klasy `String` i przypisz do każdego jej elementu odpowiedni obiekt `String`. Wypisz zawartość tablicy w pętli `for` (1).

Ćwiczenie 17. Utwórz klasę z konstruktorem, który przyjmuje argument typu `String`. W czasie konstrukcji konstruktor ma wypisać wartość argumentu na wyjściu. Utwórz tablicę referencji do obiektów tej klasy, ale nie twórz samych obiektów. Po uruchomieniu programu sprawdź, czy program wypisuje komunikaty sygnalizujące inicjalizowanie obiektów w konstruktorze Twojej klasy (2).

Ćwiczenie 18. Uzupełnij poprzednie ćwiczenie, przypisując obiekty do referencji zebranych w tablicy (1).

Zmienne listy argumentów

Druga postać inicjalizacji tablicy zapewnia wygodną składnię do tworzenia i wywołania metod, które mogą pozwolić na uzyskanie tego samego efektu co lista *argumentów zmiennej długości* (znana jako *varargs*) w języku C. Mogą one obejmować nieokreśloną liczbę argumentów i to dowolnych typów. Ponieważ wszystkie klasy są wywiedzione ze wspólnej klasy bazowej `Object` (ten temat poznasz dokładniej w miarę czytania), to można stworzyć metodę pobierającą tablicę obiektów i wywołać ją tak:

```
//: initialization/VarArgs.java
// Składnia inicjalizacji tablicy w roli mechanizmu
// zmiennej liczby argumentów metod.
```

```
class A {}

public class VarArgs {
  static void printArray(Object[] args) {
    for(Object obj : args)
      System.out.print(obj + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    printArray(new Object[]{
      new Integer(47), new Float(3.14), new Double(11.11)
    });
    printArray(new Object[]{"raz", "dwa", "trzy" });
    printArray(new Object[]{new A(), new A(), new A()});
  }
} /* Output: (Sample)
47 3.14 11.11
raz dwa trzy
A@1a46e30 A@3e25a5 A@19821f
*///:~
```

Jak widać, do metody `print()` przekazywana jest tablica obiektów `Object`, a metoda pobiera i wyświetla kolejno każdy z nich. W przypadku standardowych klas Javy wyniki generowane w ten sposób są całkiem sensowne, jednak obiekty klas zdefiniowanych w powyższym przykładzie są przedstawiane jako nazwa klasy, znak `@` i liczba szesnastkowa. A zatem domyślnie wyświetlana jest nazwa klasy oraz adres obiektu (ten domyślny sposób prezentacji można zmienić, definiując we własnych klasach metodę `toString()`, którą przedstawię w dalszej części książki).

W kodzie napisanym pod kątem wersji poprzedzających Java SE5 często widuje się powyższe techniki konstruowania zmiennych list argumentów. Ale wraz z SE5 doczekaliśmy się wreszcie dodania tej możliwości do języka: teraz zmienną listę argumentów definiuje się za pośrednictwem wielokropka, jak w poniższej metodzie `printArray()`:

```
//: initialization/NewVarArgs.java
// Nowa składnia zmiennej listy argumentów.

public class NewVarArgs {
  static void printArray(Object... args) {
    for(Object obj : args)
      System.out.print(obj + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    // Można przekazywać argumenty z osobna:
    printArray(new Integer(47), new Float(3.14),
      new Double(11.11));
    printArray(47, 3.14F, 11.11);
    printArray("raz", "dwa", "trzy");
    printArray(new A(), new A(), new A());
    // albo w tablicy:
    printArray((Object[])new Integer[]{ 1, 2, 3, 4 });
    printArray(); // Pusta lista też jest dozwolona
  }
} /* Output: (75% match)
47 3.14 11.11
```

```
47 3.14 11.11
raz dwa trzy
A@1bab50a A@c3c749 A@150bd4d
1 2 3 4
*///:~
```

Dzięki zmiennym listom argumentów nie trzeba już jawnie korzystać ze składni inicjalizacji tablicowej — kompilator samodzielnie zmontuje odpowiednią tablicę. Wciąż przekazywanie odbywa się właśnie w tablicy, dlatego też w metodzie `printArray()` można skorzystać ze składni foreach. Jednak mamy tu do czynienia z czymś więcej niż tylko automatyczną konwersją listy elementów na postać tablicy. Zwróć uwagę na przedostatni wiersz programu, gdzie tablica obiektów `Integer` (utworzonych za pomocą mechanizmu pakowania) jest rzutowana na tablicę obiektów `Object` (w celu pozbycia się ostrzeżenia ze strony kompilatora), a następnie przekazywana do metody `printArray()`. Najwyraźniej kompilator rozpoznaje tablicę i nie podejmuje konwersji. Więc jeśli masz zestaw wartości, możemy je przekazać jako listę, a jeżeli dysponujemy już gotową tablicą, możemy przekazać tę tablicę.

Ostatni wiersz programu pokazuje, że można w ten sposób skonstruować również tablicę pustą, to znaczy wywołać metodę z pustą listą argumentów. Przydaje się to tam, gdzie na końcu listy argumentów występują argumenty opcjonalne:

```
// : initialization/OptionalTrailingArguments.java

public class OptionalTrailingArguments {
  static void f(int required, String... trailing) {
    System.out.print("wymagany: " + required + " ");
    for(String s : trailing)
      System.out.print(s + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    f(1, "raz");
    f(2, "dwa", "trzy");
    f(0);
  }
} /* Output:
wymagany: 1 raz
wymagany: 2 dwa trzy
wymagany: 0
*///:~
```

Pokazuje to też, że zmienne listy argumentów można stosować z typami innymi niż `Object`. W omawianym przykładzie wszystkie argumenty mają być obiektami typu `String`. Listy zmiennych argumentów mogą używać dowolnych typów, z typami podstawowymi włącznie. Kolejny przykład pokazuje też, że taka lista staje się tablicą, a jeśli jest pusta — tablicą o zerowym rozmiarze:

```
// : initialization/VarargType.java

public class VarargType {
  static void f(Character... args) {
    System.out.print(args.getClass());
    System.out.println(" rozmiar " + args.length);
  }
```

```
  static void g(int... args) {
    System.out.print(args.getClass());
    System.out.println(" rozmiar " + args.length);
  }
  public static void main(String[] args) {
    f('a');
    f();
    g(1);
    g();
    System.out.println("int[]: " + new int[0].getClass());
  }
} /* Output:
class [Ljava.lang.Character; rozmiar 1
class [Ljava.lang.Character; rozmiar 0
class [I rozmiar 1
class [I rozmiar 0
int[]: class [I
*///:~
```

Metoda `getClass()` należy do klasy `Object` i zostanie szczegółowo przedstawiona w rozdziale „Informacje o typach". Zwraca ona nazwę klasy obiektu; wypisywanie tej nazwy pozwala na podejrzenie zakodowanego ciągu reprezentującego typ klasy. Symbol `[` na początku oznacza, że typ jest tablicą elementów typu nazwanego dalej. Symbol `I` oznacza typ podstawowy `int`; aby to potwierdzić, utworzyłem jawnie w ostatnim wierszu tablicę takich elementów i wypisałem jej typ. Dowodzi to przy okazji, że przy montowaniu tablicy dla zmiennej listy argumentów typu podstawowego nie odbywa się pakowanie elementów tablicy — do metody przekazywana jest tablica wartości podstawowych.

Zmienne listy argumentów działają zgodnie również z mechanizmem automatycznego pakowania w obiekty. Oto przykład:

```
//: initialization/AutoboxingVarargs.java

public class AutoboxingVarargs {
  public static void f(Integer... args) {
    for(Integer i : args)
      System.out.print(i + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    f(new Integer(1), new Integer(2));
    f(4, 5, 6, 7, 8, 9);
    f(10, new Integer(11), 12);
  }
} /* Output:
1 2
4 5 6 7 8 9
10 11 12
*///:~
```

Zauważ, że w pojedynczej liście argumentów można mieszać typy, a mechanizm pakowania wybiórczo dokona promocji typu `int` na typ `Integer`.

Zmienne listy argumentów komplikują proces przeciążania, choć z pozoru wydają się z nim współgrać:

```
//: initialization/OverloadingVarargs.java

public class OverloadingVarargs {
  static void f(Character... args) {
    System.out.print("pierwszy");
    for(Character c : args)
      System.out.print(" " + c);
    System.out.println();
  }
  static void f(Integer... args) {
    System.out.print("drugi");
    for(Integer i : args)
      System.out.print(" " + i);
    System.out.println();
  }
  static void f(Long... args) {
    System.out.println("trzeci");
  }
  public static void main(String[] args) {
    f('a', 'b', 'c');
    f(1);
    f(2, 1);
    f(0);
    f(0L);
    //! f(); // Niejednoznaczność uniemożliwi kompilację
  }
} /* Output:
pierwszy a b c
drugi 1
drugi 2 1
drugi 0
trzeci
*///:~
```

W każdym przypadku kompilator używa mechanizmu pakowania w celu dopasowania wywołania metody przeciążonej, wybierając wersję najlepiej pasującą.

Ale przy wywołaniu `f()` bez argumentów, kompilator nie będzie wiedział, o które wywołanie chodzi. Choć taki błąd jest zrozumiały, dla programisty-klienta może być niespodzianką.

Problem można próbować rozwiązać uzupełniając listę argumentów jednej z metod argumentem wymaganym, który mógłby różnicować przeciążone wersje metody:

```
//: initialization/OverloadingVarargs2.java
// {CompileTimeError} (Nie skompiluje się)

public class OverloadingVarargs2 {
  static void f(float i, Character... args) {
    System.out.println("pierwszy");
  }
  static void f(Character... args) {
    System.out.print("drugi");
  }
  public static void main(String[] args) {
    f(1, 'a');
    f('a', 'b');
  }
} ///:~
```

Znacznik komentarza {CompileError} wyłącza plik z kompilacji automatycznej realizowanej za pomocą programu Ant. Ręczna kompilacja pliku spowoduje pojawienie się komunikatu o błędzie:

> *reference to f is ambiguous, both method f(float,java.lang.Character...) in OverloadingVarargs2 and method f(java.lang.Character...) in OverloadingVarargs2 match*

Całość zadziała, kiedy obie metody zostaną wyposażone w argument niebędący zmienną listą argumentów:

```
//: initialization/OverloadingVarargs3.java

public class OverloadingVarargs3 {
  static void f(float i, Character... args) {
    System.out.println("pierwszy");
  }
  static void f(char c, Character... args) {
    System.out.println("drugi");
  }
  public static void main(String[] args) {
    f(1, 'a');
    f('a', 'b');
  }
} /* Output:
pierwszy
drugi
*///:~
```

Zasadniczo zmienną listę argumentów należy wykorzystywać tylko w jednej z przeciążonych wersji danej metody. A najlepiej wcale tego nie robić.

Ćwiczenie 19. Napisz metodę przyjmującą zmienną liczbę argumentów typu `String`. Sprawdź, czy możesz do niej przekazać zarówno listę obiektów `String` oddzielanych przecinkami, jak i tablicę `String[]` (2).

Ćwiczenie 20. Napisz metodę `main()`, która zmieni zwyczajową składnię `main()` przez użycie zmiennej listy argumentów. Wypisz wszystkie elementy przekazanej do `main()` tablicy `args`. Przetestuj działanie programu z różną liczbą argumentów wywołania (1).

Typy wyliczeniowe

Kolejnym uzupełnieniem języka w wydaniu SE5 jest słowo kluczowe `enum`, znacznie ułatwiające życie programisty, kiedy ten musi zgrupować i grupowo używać *wyliczeń*. W przeszłości trzeba było w ich miejsce tworzyć zbiory stałych wartości całkowitych, ale nawet obecność takich definicji nijak nie ograniczała zbioru wartości wyliczenia, przez co ich używanie było trudniejsze i ryzykowne. Typy wyliczeniowe są na tyle powszechne, że stosuje się je od zawsze w C, C++ i wielu innych językach programowania. Programiści Javy przed pojawieniem się wersji Java SE5, chcąc używać wyliczeń, musieli zachować wielką ostrożność. Teraz również Java ma słowo `enum`, i to znacznie bardziej wszechstronne niż w C i C++. Oto prosty przykład:

```
// : initialization/Spiciness.java

public enum Spiciness {
  NOT, MILD, MEDIUM, HOT, FLAMING
} ///:~
```

Powyższy kod tworzy typ wyliczeniowy `Spiciness` z pięcioma nazwanymi wartościami. Ponieważ egzemplarze typów wyliczeniowych są stałymi, przyjęło się, że ich nazwy zapisuje się wielkimi literami (a w przypadku nazw złożonych z wielu wyrazów oddziela się je znakami podkreślenia).

Aby użyć wyliczenia, należy utworzyć referencję typu wyliczeniowego i przypisać do niej egzemplarz:

```
// : initialization/SimpleEnumUse.java

public class SimpleEnumUse {
  public static void main(String[] args) {
    Spiciness howHot = Spiciness.MEDIUM;
    System.out.println(howHot);
  }
} /* Output:
MEDIUM
*///:~
```

W reakcji na utworzenie typu wyliczeniowego kompilator podejmuje szereg użytecznych działań. Na przykład syntetyzuje metodę `toString()`, którą można wykorzystać do wypisywania nazwy egzemplarza typu wyliczeniowego — właśnie w ten sposób powyższy program generuje napis na wyjściu. Ponadto kompilator tworzy metodę `ordinal()`, informującą o miejscu deklaracji danej stałej w wyliczeniu, oraz statyczną metodę `values()`, zwracającą tablicę wartości stałych typu wyliczeniowego ułożonych w kolejności deklaracji:

```
// : initialization/EnumOrder.java

public class EnumOrder {
  public static void main(String[] args) {
    for(Spiciness s : Spiciness.values())
      System.out.println(s + ", miejsce " + s.ordinal());
  }
} /* Output:
NOT, miejsce 0
MILD, miejsce 1
MEDIUM, miejsce 2
HOT, miejsce 3
FLAMING, miejsce 4
*///:~
```

Choć wyliczenie zdaje się stanowić nowy typ danych, to tak naprawdę słowo `enum` wymusza jedynie na kompilatorze wygenerowanie klasy dla definiowanego wyliczenia; można więc pod pewnymi względami traktować wyliczenie `enum` jako klasę. W rzeczy samej są to klasy i nawet posiadają metody.

Szczególnie miłą cechą wyliczeń jest możliwość wygodnego ich użycia w instrukcji wielokrotnego wyboru:

```
// : initialization/Burrito.java

public class Burrito {
  Spiciness degree;
  public Burrito(Spiciness degree) { this.degree = degree;}
  public void describe() {
    System.out.print("To burrito jest ");
    switch(degree) {
      case NOT:    System.out.println("łagodne.");
                   break;
      case MILD:
      case MEDIUM: System.out.println("trochę pikantne.");
                   break;
      case HOT:
      case FLAMING:
      default:     System.out.println("trochę za ostre.");
    }
  }
  public static void main(String[] args) {
    Burrito
      plain = new Burrito(Spiciness.NOT),
      greenChile = new Burrito(Spiciness.MEDIUM),
      jalapeno = new Burrito(Spiciness.HOT);
    plain.describe();
    greenChile.describe();
    jalapeno.describe();
  }
} /* Output:
To burrito jest łagodne.
To burrito jest trochę pikantne.
To burrito jest nieco za ostre.
*///:~
```

Ponieważ instrukcja wielokrotnego wyboru ma różnicować wykonanie programu przez dopasowywanie wartości z pewnego ograniczonego zbioru, taki zbiór świetnie nadaje się na wyliczenie. Dzięki temu można stosować w blokach case nazwy elementów wyliczenia, co niejednokrotnie zwiększa czytelność kodu.

Wyliczenia stosuje się tak, jakby był to kolejny sposób definiowania typu danych, a potem po prostu stosuje. O to właśnie chodzi — aby nie trzeba było się nad nimi zbyt wiele biedzić. Przed wprowadzeniem słowa kluczowego enum w wydaniu SE5 języka powołanie do życia bezpiecznego w użyciu odpowiednika typu wyliczeniowego wymagało pewnego wysiłku.

Tyle wystarczy, żeby ogarnąć wyliczenia i zacząć je stosować; do tematu powrócę w poświęconym mu w całości rozdziale „Typy wyliczeniowe".

Ćwiczenie 21. Utwórz wyliczenie reprezentujące sześć najmniej wartościowych banknotów. Przejrzyj wartości wyliczenia zwracane metodą values() i wypisz nazwy elementów wyliczenia oraz ich pozycje (ordinal()) (1).

Ćwiczenie 22. Napisz instrukcję switch obsługującą typ wyliczeniowy z poprzedniego ćwiczenia. W każdym bloku case wypisz opis danego banknotu (2).

Podsumowanie

Ten pozornie zawiły mechanizm inicjalizacji — w postaci konstruktora — powinien stanowić istotną wskazówkę na temat roli i znaczenia inicjalizacji w języku. Gdy Bjarne Stroustrup projektował język C++, to jedną z pierwszych obserwacji, które poczynił na temat produktywności w C, była uwaga, że niewłaściwa inicjalizacja zmiennych powoduje znaczną część problemów programistycznych. Błędy tego rodzaju są trudne do odnalezienia. Podobnie wygląda sprawa niewłaściwego sprzątania. Ponieważ konstruktory pozwalają *zagwarantować* właściwą inicjalizację i sprzątanie (kompilator nie pozwoli na stworzenie obiektu bez poprawnego wywołania konstruktora), zyskujemy pełną kontrolę i bezpieczeństwo.

W C++ dosyć istotna jest destrukcja, ponieważ obiekty stworzone poprzez `new` muszą zostać jawnie zniszczone. W Javie odśmiecacz pamięci automatycznie zwalnia pamięć wszystkich obiektów, zatem równoważna metoda sprzątająca w większości przypadków nie jest potrzebna (a jeśli jest, należy ją zdefiniować samemu). Gdy nie potrzebujemy zachowania podobnego do destruktora, odśmiecacz pamięci wspaniale upraszcza programowanie i dodaje, jakże potrzebne, bezpieczeństwo w zarządzaniu pamięcią. Niektóre odśmiecacze pamięci mogą nawet czyścić inne zasoby, jak choćby grafikę i uchwyty do plików. Jednak odśmiecacz pamięci zwiększa koszty działania, które są trudne do oszacowania z uwagi na ogólne spowolnienie interpreterów Javy. Choć efektywność działania Javy uległa znaczącej poprawie, problem szybkości odcisnął swe piętno na wykorzystaniu Javy do rozwiązywania pewnych typów zagadnień programistycznych.

Ze względu na gwarancję poprawnego stworzenia *wszystkich* obiektów, temat konstruktorów jest obszerniejszy niż opis zamieszczony w tym rozdziale. Przy tworzeniu nowej klasy za pomocą *dziedziczenia* lub *kompozycji* nadal musi być spełniony warunek poprawnej konstrukcji wszystkich obiektów, i — jak się okaże — wymaga to wprowadzenia dodatkowej składni. Kompozycja, dziedziczenie oraz ich wpływ na konstruktory są przedstawione w dalszych rozdziałach.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 6.

Kontrola dostępu

Kontrola dostępu (tudzież ukrywanie informacji) sprowadza się do poprawiania tego, co się nie udało za pierwszym razem.

Wszyscy piszący — w tej liczbie programiści tworzący programy — wiedzą, że dzieło nigdy nie jest dobre za pierwszym razem, a często trzeba je przepisywać wielokrotnie. Jeśli odłożymy na jakiś czas kawałek kodu, dając mu szansę „przeschnąć", po powrocie do niego z pewnością znajdziemy jakieś niedoskonałości. Stąd bierze się *refaktoryzacja* kodu, która polega na jego przepisywaniu tak, aby był czytelniejszy, bardziej zrozumiały, łatwiejszy do ogarnięcia i przyszłej konserwacji[1].

Owa żądza poprawiania i ulepszania kodu kłóci się jednak z pewnym oczekiwaniem klientów (*programistów użytkujących* dany kod), którzy chcieliby móc założyć, że pewne aspekty kodu pozostaną niezmienne. Mamy więc konflikt interesów: twórcy chcieliby zmieniać, odbiorcy — zachowywać w stanie niezmienionym. Dlatego Pierwszą sprawą, braną pod uwagę przy programowaniu obiektowym, jest „oddzielenie rzeczy, które się zmieniają, od rzeczy pozostających bez zmian".

Jest to szczególnie istotne w przypadku bibliotek. Klienci takiej biblioteki muszą polegać na części, którą wykorzystują, i nie obawiać się, że będą musieli przepisywać kod, jeżeli pojawi się nowa wersja. Z drugiej strony, twórca biblioteki musi mieć swobodę jej modyfikowania i ulepszania przy zachowaniu pewności, że zmiany te nie będą miały wpływu na kod klientów.

Można to osiągnąć poprzez ustalenie konwencji. Na przykład, programista biblioteki musi zgodzić się na nieusuwanie istniejących metod przy modyfikowaniu klas tej biblioteki, ponieważ uszkodziłoby to kod klientów. Sytuacja odwrotna jest jednak trudniejsza. W przypadku pól klasy twórca biblioteki nie może wiedzieć, które z nich były używane przez klientów. Dotyczy to także metod stanowiących jedynie część implementacji klasy, nieprzewidzianych do bezpośredniego wykorzystania przez programistę-klienta.

---

[1] Polecam *Refactoring: Improving the Design of Existing Code* Martina Fowlera i innych (Addison-Wesley, 1999). Okazjonalnie pojawiają się głosy przeciwko refaktoryzacji sugerujące, że kod działający to kod perfekcyjny i dłubanie w nim to czysta strata czasu. Problem w tym podejściu polega na tym, że lwia część nakładów czasowych i finansowych na projekty nie idzie wcale na ich powstanie, ale na utrzymanie. Zwiększenie czytelności i przejrzystości kodu przekłada się więc na grube dolary.

Co zrobić, gdy twórca biblioteki chciałby wyrzucić starą implementację i zastosować nową? Zmiana któregokolwiek ze wspomnianych składników klas może uszkodzić kod ich użytkownika. Na twórcę biblioteki nałożony jest „gorset" niepozwalający mu niczego zmienić.

W celu rozwiązania tego problemu Java dostarcza *modyfikatory dostępu* (ang. *access modifiers*). Umożliwiają one twórcy biblioteki zaznaczenie, co ma być dostępne dla klienta, a co nie. Poziomy dostępu w porządku od największego do najmniejszego to: `public`, `protected`, pakietowy (który nie posiada słowa kluczowego) i `private`. Na podstawie poprzedniego akapitu można wysnuć wniosek, że projektant biblioteki powinien wszystko, co się da, uczynić prywatnym, a eksponować jedynie to, co powinno być jego zdaniem wykorzystywane przez klientów. Jest to prawda, mimo że może być to sprzeczne z intuicją osób programujących w innych językach (szczególnie w C) i będących przyzwyczajonymi do nieograniczonego dostępu do wszystkiego. Przed końcem tego rozdziału powinieneś być przekonany o wartości kontroli dostępu.

Koncepcja biblioteki komponentów i kontroli dostępu do nich nie jest jednak kompletna. Pozostaje pytanie: w jaki sposób komponenty są łączone w spójne jednostki biblioteczne? W Javie jest to kontrolowane przez słowo kluczowe `package` (pakiet), na modyfikatory dostępu ma zaś wpływ to, czy klasy znajdują się w tym samym pakiecie czy też w różnych. Zatem na początek dowiemy się, w jaki sposób komponenty bibliotek są umieszczane w pakietach. Dzięki temu możliwe będzie pełne zrozumienie modyfikatorów dostępu.

Pakiet — jednostka biblioteczna

Pakiet to grupa klas zebranych razem we wspólnej *przestrzeni nazw*.

Istnieje na przykład biblioteka narzędziowa wchodząca w skład standardowej dystrybucji Javy, skompletowana w przestrzeni nazw `java.util`. Jedną z klas tej biblioteki jest klasa `ArrayList`. Klasę tę możemy wykorzystywać za pośrednictwem pełnej nazwy: `java.util.ArrayList`.

```
//: access/FullQualification.java

public class FullQualification {
  public static void main(String[] args) {
    java.util.ArrayList list = new java.util.ArrayList();
  }
} ///:~
```

Szybko stałoby się to uciążliwe, dlatego lepiej używać słowa kluczowego `import`. Jeśli zachodzi potrzeba importowania pojedynczej klasy, należy wymienić tę klasę w instrukcji `import`:

```
//: access/SingleImport.java
import java.util.ArrayList;

public class SingleImport {
  public static void main(String[] args) {
    ArrayList list = new java.util.ArrayList();
  }
} ///:~
```

Teraz możemy używać `ArrayList` bez dodatkowych kwalifikatorów, jednakże inne klasy pakietu `java.util` nie są dostępne. Aby zaimportować całość biblioteki, należy skorzystać z symbolu wieloznacznego — * — powszechnie wykorzystywanego w poprzednich przykładach z tej książki:

```
import java.util.*;
```

Uzasadnieniem tego całego importowania jest dostarczenie mechanizmu zarządzania przestrzeniami nazw. Nazwy składników wszystkich klas są od siebie odizolowane. Metoda `f()` klasy `A` nie wejdzie w konflikt z metodą `f()` klasy `B`, nawet jeśli obie metody mają taką samą sygnaturę. Co jednak z nazwami klas? Przypuśćmy, że tworzymy klasę `Stos`, instalowaną na maszynie posiadającej już `Stos` napisany przez kogoś innego. Ten potencjalny konflikt jest przyczyną, dla której w Javie istotne jest posiadanie całkowitej kontroli nad przestrzeniami nazw oraz tworzenie unikatowej nazwy dla każdej klasy.

Do tej pory większość przykładów w tej książce składała się z jednego pliku i była zaprojektowana do użytku lokalnego — nie musieliśmy przejmować się nazwami pakietów. Ale klasy z tych przykładów były umieszczane w pakiecie — pakiecie „nienazwanym", inaczej *pakiecie domyślnym*. Jest to z pewnością jakieś rozwiązanie i ze względu na swą prostotę będzie stosowane w dalszej części książki, gdziekolwiek będzie to możliwe. Jeżeli jednak planujemy tworzenie bibliotek lub programów przyjaznych dla innych programów Javy na tej samej maszynie, musimy zapobiec konfliktom nazw klas.

Gdy tworzymy plik z kodem źródłowym Javy, jest on powszechnie nazywany *jednostką kompilacji* (czasami również *jednostką translacji*). Nazwa każdej jednostki kompilacji musi mieć rozszerzenie *.java*. Wewnątrz takiej jednostki może znajdować się publiczna klasa, której nazwa musi być taka sama, jak nazwa pliku (włączając w to wielkość liter, wyłączając natomiast rozszerzenie *.java*). W każdej jednostce kompilacji może znajdować się tylko *jedna* klasa publiczna, w przeciwnym razie kompilator zaprotestuje. Ewentualne pozostałe klasy wchodzące w skład jednostki kompilacji są ukryte przed światem zewnętrznym, ponieważ nie są publiczne. Stanowią one „klasy wspierające" zamieszczonej głównej klasy publicznej.

Organizacja kodu

Kompilując plik typu *.java*, otrzymujemy plik wynikowy o dokładnie takiej samej nazwie, lecz z rozszerzeniem *.class dla każdej klasy* z pliku *.java*. Możliwe jest więc uzyskanie całkiem sporej liczby plików *.class* z niewielu plików *.java*. Jeżeli pracowałeś kiedyś z językiem kompilowanym, możesz być przyzwyczajony do kompilatora produkującego formy pośrednie (zwykle pliki typu „obj"), które następnie łączone są przy użyciu linkera (w celu wyprodukowania pliku wykonywalnego) lub bibliotekarza (w celu utworzenia biblioteki). Java nie działa w ten sposób. Działający program to zbiór plików typu *.class*, które mogą zostać spakowane i skompresowane do pliku typu JAR (z wykorzystaniem programu archiwizującego *jar*). Interpreter Javy odpowiada za znajdowanie, ładowanie i interpretowanie tych plików[2].

[2] Żaden element Javy nie wymusza używania interpretera. Istnieją kompilatory Javy produkujące pojedynczy plik wykonywalny zawierający kod natywny danej platformy.

Biblioteka również stanowi zespół plików zawierających klasy. Każdy plik zawiera jedną klasę publiczną (nie musi zawierać takiej klasy, ale zwykle tak się dzieje), a zatem na każdy plik przypada jeden komponent. Jeżeli chcemy zaznaczyć, że wszystkie te komponenty (znajdujące się w oddzielnych plikach *.java* i *.class*) stanowią jedną całość, używamy słowa kluczowego package.

Jeśli już stosujemy instrukcję package, to *musi* ona być pierwszą (poza ewentualnymi komentarzami) instrukcją w pliku. Gdy piszemy:

```
package access;
```

tym samym zaznaczamy, że ta jednostka kompilacji znajduje się pod parasolem nazwy access, a zatem każdy, kto chciałby wykorzystać zawarte w niej nazwy, musi albo wyspecyfikować pełną nazwę, albo użyć słowa kluczowego import w połączeniu z access (na jeden z podanych wcześniej sposobów). Zauważmy, że konwencja nazywania pakietów w Javie polega na używaniu wyłącznie małych liter, nawet jeśli nazwa składa się z kilku słów.

Przypuśćmy np., że plik nazywa się MyClass.java. Znaczy to, że może znajdować się w nim jedna i tylko jedna klasa publiczna, a jej nazwą musi być MyClass (wielkość liter ma znaczenie):

```
//: access/mypackage/MyClass.java
package access.mypackage;

public class MyClass {
  // ...
} ///:~
```

Teraz, jeżeli ktoś chce wykorzystać klasę MyClass lub jakąkolwiek inną klasę publiczną z pakietu access, musi użyć słowa kluczowego import, aby uczynić dostępnymi nazwy zawarte w pakiecie access. Inną możliwością jest użycie nazwy z pełnym kwalifikatorem:

```
//: access/QualifiedMyClass.java

public class QualifiedMyClass {
  public static void main(String[] args) {
    access.mypackage.MyClass m =
      new access.mypackage.MyClass();
  }
} ///:~
```

Słowo kluczowe import czyni ten kod znacznie czystszym:

```
//: access/ImportedMyClass.java
import access.mypackage.*;

public class ImportedMyClass {
  public static void main(String[] args) {
    MyClass m = new MyClass();
  }
} ///:~
```

Warto zapamiętać, że dzięki słowom kluczowym package i import projektanci bibliotek mogą podzielić pojedynczą globalną przestrzeń nazw, aby zapobiec ich konfliktom bez względu na to, jak wielu ludzi przyłączy się do Internetu i zacznie tworzyć klasy Javy.

Tworzenie unikatowych nazw pakietów

Można zauważyć, że ponieważ pakiet nigdy nie zostaje „spakowany" do pojedynczego pliku, lecz może się składać z wielu plików *.class*, zaczyna się powoli robić bałagan. Aby temu zapobiec, logiczne wydaje się umieszczenie wszystkich plików *.class*, składających się na dany pakiet, w jednym katalogu — a zatem wykorzystanie hierarchicznej struktury plików dostarczanej przez system operacyjny. Jest to jeden ze sposobów, w jaki Java radzi sobie z problemem bałaganu. Inny sposób poznamy później, przy omawianiu narzędzia *jar*.

Zebranie plików pakietu w pojedynczym podkatalogu rozwiązuje także dwa inne problemy: tworzenie unikatowych nazw pakietów oraz odnajdywanie tych klas, które mogłyby być ukryte gdzieś w strukturze katalogów. Osiągane jest to poprzez zakodowanie ścieżki dostępu do pliku *.class* w nazwie pakietu. Zgodnie z konwencją przyjmuje się, że pierwszą część nazwy pakietu powinna stanowić odwrócona nazwa domeny internetowej twórcy klasy. Ponieważ unikatowość nazw domen internetowych jest gwarantowana, zatem *jeżeli* przestrzegamy konwencji, gwarantowana jest również unikatowość nazwy naszego pakietu, i nie ma możliwości powstania konfliktu (przynajmniej do chwili, gdy utracimy nazwę domeny na rzecz kogoś, kto zacznie pisać kod Javy z takimi samymi nazwami ścieżek jak nasze). Oczywiście jeżeli nie posiadamy własnej domeny, jesteśmy zmuszeni do sfabrykowania jakiejś mało prawdopodobnej kombinacji (np. własnego imienia i nazwiska) w celu nazywania pakietów. Jeżeli decydujemy się na rozpoczęcie publikowania kodu w Javie, warto zdobyć się na stosunkowo niewielki wysiłek pozyskania nazwy domeny.

Drugą częścią sztuczki jest odzyskiwanie nazwy katalogu na naszej maszynie z nazwy pakietu, aby działający program Javy, w razie konieczności załadowania pliku *.class*, mógł zlokalizować katalog, w którym znajduje się ten plik.

Interpreter Javy działa w następujący sposób. Na wstępie odnajduje zmienną środowiskową `CLASSPATH`[3] (ustawianą przez system operacyjny, czasami przez program instalujący Javę lub narzędzie wykorzystujące Javę na naszej maszynie). Zmienna `CLASSPATH` zawiera jeden lub więcej katalogów używanych jako korzenie przy wyszukiwaniu plików *.class*. Zaczynając od takiego korzenia, interpreter zamieni wszystkie kropki w nazwie pakietu na znaki ukośnika w celu utworzenia ścieżki dostępu (a zatem pakiet `foo.bar.baz` zostanie zamieniony na `foo\bar\baz` lub `foo/bar/baz` albo, być może, jeszcze inny w zależności od naszego systemu operacyjnego). Rezultat zostanie następnie dołączony do kolejnych katalogów z `CLASSPATH`. W taki sposób wyznaczone zostają miejsca, w których poszukuje się pliku *.class* o nazwie odpowiadającej nazwie klasy, której instancję chcemy stworzyć (interpreter przeszukuje również pewne standardowe katalogi względem miejsca, w którym on sam został zainstalowany).

Aby to zrozumieć, rozważmy nazwę domeny autora, tj. *MindView.net*. Poprzez jej odwrócenie otrzymujemy `net.mindview`, unikatową globalną nazwę dla moich klas (rozszerzenia com, edu, org itd. były kiedyś pisane wielkimi literami, jednak w Javie 2 zostało to

[3] Odwołując się do zmiennych środowiskowych, ich nazwy będę zapisywać wielkimi literami (np.: `CLASSPATH`).

zmienione tak, aby w nazwach pakietów występowały wyłącznie małe litery). Możliwe jest dalsze podzielenie tej przestrzeni nazw, kiedy chcę np. utworzyć bibliotekę `simple`, przez co powstanie np.:

```
package net.mindview.simple;
```

Taka nazwa pakietu może zostać teraz użyta jako parasol nazw dla następujących dwóch plików:

```
//: net/mindview/simple/Vector.java
// Tworzenie pakietu.
package net.mindview.simple;

public class Vector {
  public Vector() {
    System.out.println("net.mindview.simple.Vector");
  }
} ///:~
```

Jak już wspomniałem, instrukcja `package` musi być pierwszym nie będącym komentarzem fragmentem kodu w pliku. Drugi z plików wygląda bardzo podobnie:

```
//: net/mindview/simple/List.java
// Tworzenie pakietu.
package net.mindview.simple;

public class List {
  public List() {
    System.out.println("net.mindview.simple.List");
  }
} ///:~
```

Oba pliki zostały umieszczone w systemie autora w podkatalogu:

```
C:\DOC\JavaT\net\mindview\simple
```

(Zauważ, że pierwszy wiersz każdego pliku prezentowanego w tej książce wymienia lokalizację pliku w drzewie katalogów kodu źródłowego — wiersz ten jest wykorzystywany przez zautomatyzowane narzędzie wyodrębniania kodu, które stosowałem przy pisaniu książki.)

Cofając się nieco, możemy odnaleźć nazwę pakietu *net.mindview.simple*, co jednak z pierwszą częścią ścieżki? Tym zajmuje się zmienna środowiskowa `CLASSPATH`, która na maszynie autora ma wartość:

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

Możemy zaobserwować, że zmienna `CLASSPATH` może zawierać kilka alternatywnych ścieżek poszukiwań.

Pewne odstępstwo dotyczy plików typu JAR. W zmiennej `CLASSPATH` musimy umieścić nazwę pliku JAR, a nie tylko ścieżkę do miejsca, gdzie się znajduje. A zatem dla pliku nazywającego się *grape.jar* mamy:

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

Kiedy zmienna CLASSPATH jest już poprawnie ustawiona, wtedy następujący plik może być umieszczony w dowolnym katalogu:

```
//: access/LibTest.java
// Użycie biblioteki.
import net.mindview.simple.*;

public class LibTest {
  public static void main(String[] args) {
    Vector v = new Vector();
    List l = new List();
  }
} /* Output:
net.mindview.simple.Vector
net.mindview.simple.List
*///:~
```

Gdy kompilator napotyka instrukcję `import`, rozpocznie przeszukiwanie katalogów wymienionych w zmiennej CLASSPATH w poszukiwaniu podkatalogu *net\mindview\simple*, a następnie skompilowanych plików o odpowiednich nazwach (*Vector.class* dla klasy `Vector` i *List.class* dla klasy `List`). Należy zauważyć, że zarówno same klasy, jak i potrzebne ich metody muszą być publiczne.

Ustawianie zmiennej CLASSPATH było tak trudne dla początkujących użytkowników Javy (także dla autora), że firma Sun uczyniła następne wersje JDK nieco zmyślniejszymi. Po ich zainstalowaniu odkryjemy, że skompilowanie i uruchomienie prostych programów jest możliwe bez ustawiania CLASSPATH. W celu skompilowania i uruchomienia kodu źródłowego (dostępnego pod adresem *www.MindView.net*) konieczne będzie dodanie do zmiennej CLASSPATH ścieżki dostępu do głównego katalogu zawierającego kody przykładów.

Ćwiczenie 1. Utwórz klasę w pakiecie. Utwórz egzemplarz tej klasy poza pakietem (1).

Kolizje

Co się dzieje, gdy program importuje z użyciem znaku * dwie biblioteki zawierające te same nazwy? Na przykład, przypuśćmy, że program zawiera instrukcje:

```
import net.mindview.simple.*;
import java.util.*;
```

Ponieważ pakiet `java.util.*` również zawiera klasę `Vector`, powoduje to potencjalną kolizję. Jednak wszystko jest w porządku, dopóki nie napiszemy kodu, który rzeczywiście spowoduje kolizję — jest to dobre rozwiązanie, ponieważ w przeciwnym razie zmuszeni bylibyśmy tworzyć rozbudowany kod, aby uniemożliwić kolizje, które nigdy by się nie zdarzyły.

Kolizja *ma* miejsce, gdy spróbujemy teraz stworzyć `Vector`:

```
Vector v = new Vector();
```

Do której klasy `Vector` to się odnosi? Kompilator nie może tego wiedzieć, podobnie jak czytający kod. Zatem kompilator protestuje i zmusza do jawności. Jeżeli chcemy, na przykład, stworzyć standardowy `Vector` Javy, musimy napisać:

```
java.util.Vector v = new java.util.Vector();
```

Ponieważ powyższa instrukcja (wraz z wartością CLASSPATH) dokładnie określa położenie klasy `Vector`, nie ma zatem potrzeby poprzedzać jej konstrukcją import `java.util.*`, chyba że używamy jeszcze innych elementów pakietu `java.util`.

Alternatywną metodą zapobiegania kolizji byłby import pojedynczej klasy — dopóki nie użylibyśmy kolidujących nazw w tym samym programie (wtedy znów trzeba by się uciec do pełnej kwalifikacji nazw).

Ćwiczenie 2. Zbierz fragmenty kodu z tego punktu w programie i sprawdź, czy faktycznie dojdzie do kolizji nazw (1).

Własna biblioteka narzędziowa

Uzbrojeni w tę wiedzę możemy stworzyć własne biblioteki pomocnicze w celu zredukowania lub wyeliminowania powtarzania kodu. Można, na przykład, rozważyć wprowadzenie innej nazwy dla `System.out.println()` w celu ograniczenia ilości pisanego kodu. Mogłaby ona być częścią klasy `Print`, tak aby dało się importować ją czytelnie z użyciem `static import`:

```
//: net/mindview/util/Print.java
// Metody wypisujące, do stosowania bez kwalifikatorów
// przy użyciu importu statycznego z Java SE5:
package net.mindview.util;
import java.io.*;

public class Print {
  // Wersja ze znakiem nowego wiersza na końcu komunikatu:
  public static void print(Object obj) {
    System.out.println(obj);
  }
  // Sam znak nowego wiersza:
  public static void print() {
    System.out.println();
  }
  // Bez znaku nowego wiersza:
  public static void printnb(Object obj) {
    System.out.print(obj);
  }
  // Nowa metoda printf() w Java SE5 (z języka C):
  public static PrintStream
  printf(String format, Object... args) {
    return System.out.printf(format, args);
  }
} ///:~
```

Tak powstały alias można wykorzystywać do wygodnego wypisywania komunikatów zarówno takich, które kończą się przejściem do nowego wiersza (`print()`), jak i bez niego (`printnb()`).

Można się domyślić, że lokalizacją powyższego pliku musi być katalog o nazwie zaczynającej się od jednej z lokalizacji podanych w zmiennej CLASSPATH, po której następuje *net/mindview/util*. Po kompilacji metody statyczne `print()` i `printnb()` będą mogły być wykorzystywane wszędzie, byleby wcześniej zostały zaimportowane statycznie:

```
//: access/PrintTest.java
// Test statycznych metod wypisujących z Print.java.
import static net.mindview.util.Print.*;

public class PrintTest {
  public static void main(String[] args) {
    print("Tylko u nas!");
    print(100);
    print(100L);
    print(3.14159);
  }
} /* Output:
Tylko u nas!
100
100
3.14159
*///:~
```

Drugim elementem omawianej biblioteki mogłaby być metoda `range()`, przedstawiona w rozdziale „Sterowanie przebiegiem wykonania", która pozwalała na wygodne stosowanie składni foreach dla prostych sekwencji liczb całkowitych:

```
//: net/mindview/util/Range.java
// Metody generujące tablice, do stosowania bez kwalifikatorów
// za pomocą importów statycznych Java SE5:
package net.mindview.util;

public class Range {
  // Generuje sekwencję [0..n)
  public static int[] range(int n) {
    int[] result = new int[n];
    for(int i = 0; i < n; i++)
      result[i] = i;
    return result;
  }
  // Generuje sekwencję [start..end)
  public static int[] range(int start, int end) {
    int sz = end - start;
    int[] result = new int[sz];
    for(int i = 0; i < sz; i++)
      result[i] = start + i;
    return result;
  }
  // Generuje sekwencję [start..end) z krokiem step
  public static int[] range(int start, int end, int step) {
    int sz = (end - start)/step;
    int[] result = new int[sz];
    for(int i = 0; i < sz; i++)
      result[i] = start + (i * step);
    return result;
  }
} ///:~
```

Od tej pory, kiedy tylko stworzysz jakieś użyteczne narzędzie, będziesz mógł dodać je do swojej własnej biblioteki. A bibliotekę `net.mindview.util` będziemy w dalszych rozdziałach uzupełniać o kolejne użyteczne elementy.

Wykorzystanie instrukcji import do zmiany zachowania

Jednym z elementów języka C, niewystępującym w Javie, jest *kompilacja warunkowa* pozwalająca na uzyskanie innego zachowania za pomocą pojedynczego przełącznika bez zmieniania reszty kodu. Powodem, dla którego rozwiązanie to nie zostało włączone do Javy, jest prawdopodobnie fakt, że w C używane było głównie do rozwiązywania problemu przenośności na różne platformy — różne fragmenty kodu włączane były w zależności od docelowej platformy. Ponieważ Java miała być z założenia automatycznie przenośna, zatem kompilacja warunkowa nie powinna być potrzebna.

Istnieją jednak inne cenne zastosowania kompilacji warunkowej. Bardzo przydatne jest jej wykorzystanie przy usuwaniu błędów. Elementy odpowiedzialne za wykrywanie błędów są włączone podczas pracy nad programem, wyłączone zaś w produkcie dostarczanym klientom. Można to osiągnąć, zmieniając importowany pakiet, a tym samym kod używany w testowej oraz finalnej wersji programu. Rozwiązanie to można zastosować do dowolnego kodu, z którego chcemy korzystać warunkowo.

Ćwiczenie 3. Utwórz dwa pakiety, `debug` i `debugoff`, zawierające identyczne klasy z metodą `debug()`. Pierwsza wersja metody ma wypisywać argument wywołania (`String`) na konsoli, druga ma nie robić zupełnie nic. Skorzystaj z importu statycznego do zaimportowania klasy do programu testowego i pokaż efekt kompilacji warunkowej (2).

Pułapka związana z pakietami

Warto zapamiętać, że zawsze, gdy tworzymy pakiet, niejawnie określamy strukturę katalogów przez nadanie mu nazwy. Pakiet *musi* znajdować się w katalogu wskazywanym przez jego nazwę i powinien być dostępny przy przeszukiwaniu katalogów za pomocą zmiennej `CLASSPATH`. Eksperymentowanie ze słowem kluczowym `package` może być na początku nieco frustrujące, ponieważ jeśli nie stosujemy się ściśle do reguły zgodności między nazwami pakietów i ścieżkami do katalogów, otrzymujemy dużo tajemniczych komunikatów czasu wykonania dotyczących niemożności odnalezienia określonych klas, nawet jeżeli klasy te znajduje się w tym samym katalogu. Po otrzymaniu takiej wiadomości można spróbować wykomentować instrukcję `package` — jeżeli po tym zabiegu wszystko zacznie działać, wtedy wiadomo już, w czym tkwi problem.

Zauważ, że kod kompilowany jest często umieszczany w innym katalogu niż kod źródłowy, ale ścieżka do kodu skompilowanego musi pozostawać w zasięgu maszyny wirtualnej, to znaczy w zasięgu zmiennej środowiskowej `CLASSPATH`.

Modyfikatory dostępu w Javie

Używając w Javie modyfikatorów dostępu: `public`, `protected` lub `private`, umieszczamy je przed definicją każdego składnika klasy, bez względu na to, czy jest to pole czy metoda.

W przypadku braku jawnego specyfikatora dostępu przyjmowany jest „dostęp pakietowy". A więc tak czy inaczej wszystko ma określony stopień dostępu. Zajmiemy się teraz różnymi ich typami, począwszy od dostępu domyślnego.

Dostęp pakietowy

W żadnym z dotychczasowych przykładów nie zastosowaliśmy żadnego specyfikatora dostępu. Domyślny stopień dostępu nie posiada swojego słowa kluczowego, określany jest jednak często jako *dostęp pakietowy* (a czasami także jako dostęp „przyjazny"). Oznacza to, że inne klasy tego samego pakietu posiadają dostęp do elementu, jednak dla klas poza pakietem wydaje się on być prywatny. Ponieważ jednostka kompilacji — plik — może należeć jedynie do jednego pakietu, zatem, dzięki zastosowaniu dostępu pakietowego, wszystkie klasy wewnątrz tej samej jednostki kompilacji są automatycznie dostępne dla siebie nawzajem.

Dostęp pakietowy pozwala na grupowanie spokrewnionych klas w pakiety, dzięki czemu mogą one z łatwością ze sobą współdziałać. Gdy programista umieszcza pewne klasy razem we wspólnym pakiecie (tym samym umożliwiając im wzajemny dostęp do zdefiniowanych w nich składowych o dostępie pakietowym), musi być „właścicielem" kodu tego pakietu. Wydaje się sensowne, że tylko kod „posiadany" przez nas powinien mieć dostęp w ramach pakietu do innego „posiadanego" przez nas kodu. Można powiedzieć, że taki dostęp nadaje sens (lub że stanowi przyczynę) grupowaniu klas w pakiety. W wielu językach sposób organizacji definicji w plikach jest całkowicie dowolny — w Javie jesteśmy zmuszeni do organizowania ich w sposób rozsądny. Dodatkowo będziemy prawdopodobnie chcieli wyłączyć klasy, które nie powinny mieć dostępu do klas zdefiniowanych w aktualnym pakiecie.

Klasa ma kontrolę nad tym, który kod ma dostęp do jej składników. Nie ma takiej możliwości, aby kod z innego pakietu pojawił się i stwierdził: „Cześć, jestem przyjacielem Boba!" i oczekiwać, że zostaną mu udostępnione chronione, „przyjacielskie" i prywatne składniki klasy Bob. Jedynymi sposobami przyznania dostępu do składnika są:

1. Uczynienie składnika publicznym (przez podanie modyfikatora `public`). Od tej chwili dostęp do niego będzie miał każdy, z dowolnego miejsca.
2. Zapewnienie dostępu do składnika w ramach pakietu poprzez rezygnację z podania modyfikatora dostępu oraz umieszczenie klas mających mieć dostęp do składnika w tym samym pakiecie.
3. Jak przekonamy się w rozdziale „Wielokrotne wykorzystanie klas" (gdzie omówimy dziedziczenie), klasa pochodna ma dostęp do składników chronionych (zadeklarowanych z modyfikatorem `protected`) oraz publicznych (modyfikator `public`). Nie ma natomiast dostępu do składników prywatnych (modyfikator `private`). Do składników „pakietowych" ma dostęp tylko wtedy, gdy obie klasy znajdują się w tym samym pakiecie. Nie musisz jednak na razie zaprzątać sobie tym głowy.
4. Dostarczenie akcesorów i modyfikatorów — metod udostępniających i zmieniających wartość składnika (w języku angielskim określane są one jako metody typu *accessor/mutator* lub *get/set*). W kategoriach programowania zorientowanego obiektowo jest to rozwiązanie najbardziej cywilizowane. Ma ono także fundamentalne znaczenie dla komponentów JavaBeans, o czym przekonamy się w rozdziale „Graficzne interfejsy użytkownika".

public: dostęp do interfejsu

Używając słowa kluczowego public, czynimy deklarację składnika następującego po nim dostępną dla każdego, w szczególności dla programisty-klienta wykorzystującego bibliotekę. Przypuśćmy, że definiujemy pakiet dessert zawierający następującą jednostkę kompilacji:

```
//: access/dessert/Cookie.java
// Tworzy bibliotekę.
package access.dessert;

public class Cookie {
  public Cookie() {
   System.out.println("Konstruktor Cookie");
  }
  void bite() { System.out.println("chrup"); }
} ///:~
```

Pamiętaj, że plik *Cookie.java* musi znajdować się w podkatalogu o nazwie *dessert*, umieszczonym w katalogu *access* (gromadzącym przykłady z tego rozdziału), który z kolei musi być podkatalogiem jednego z katalogów umieszczonych w zmiennej CLASSPATH. Błędem jest zakładanie, że Java zawsze traktuje katalog aktualny jako jeden z punktów startowych dla poszukiwań. Jeżeli nie umieścimy katalogu „." w zmiennej CLASSPATH, Java nie będzie postępować w ten sposób.

Stwórzmy teraz program wykorzystujący klasę Cookie:

```
//: access/Dinner.java
// Użycie biblioteki.
import access.dessert.*;

public class Dinner {
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //! x.bite(); // Brak dostępu
  }
} /* Output:
Konstruktor Cookie
*///:~
```

Stworzenie obiektu klasy Cookie było możliwe, ponieważ zarówno sama klasa, jak i jej konstruktor są publiczne (koncepcji klasy publicznej przyjrzymy się trochę później). Składowa (metoda) bite() jest jednak niedostępna z wnętrza pliku *Dinner.java*, ponieważ dostęp do niego jest możliwy jedynie wewnątrz pakietu dessert (o sprawdzenie praw dostępu do tego składnika zadba kompilator).

Pakiet domyślny

Może być zaskakujące, że poniższy kod kompiluje się mimo wrażenia, że łamie zasady:

```
//: access/Cake.java
// Dostęp do innej klasy w osobnej jednostce kompilacji.

class Cake {
```

```
  public static void main(String[] args) {
    Pie x = new Pie();
    x.f();
  }
} /* Output:
Pie.f()
*///:~
```

W drugim pliku, znajdującym się w tym samym katalogu, umieszczamy:

```
//: access/Pie.java
// Owa "inna" klasa.

class Pie {
  void f() { System.out.println("Pie.f()"); }
} ///:~
```

Początkowo można przypuszczać, że powyższe pliki są sobie zupełnie obce, a mimo to obiekt typu `Cake` może utworzyć obiekt `Pie` i wywołać jego metodę `f()`! (Aby możliwe było skompilowanie tego pliku, należy umieścić katalog `"."` w zmiennej `CLASSPATH`). Zwykle narzuca się przypuszczenie, że `Pie` i `f()` są składowymi o dostępie pakietowym, i przez to niedostępnymi dla klasy `Cake`. Istotnie są one dostępne w ramach pakietu — to założenie jest poprawne. Powodem, dla którego są dostępne z pliku *Cake.java*, jest to, że znajdują się w tym samym co on katalogu i nie są jawnie przypisane do żadnego pakietu. Java traktuje takie pliki jako należące do „pakietu domyślnego" dla tego katalogu, a przez to dostępne w ramach pakietu dla innych plików w tym samym katalogu.

private: nie dotykać!

Słowo kluczowe `private` oznacza, że do danego składnika klasy nie ma dostępu nikt poza jego własną klasą, z wnętrza jej metod. Inne klasy tego samego pakietu nie mają dostępu do składowych prywatnych — to tak jakbyśmy izolowali klasę nawet przed nami. Z drugiej strony, nie jest nieprawdopodobne, że pakiet będzie tworzony przez grupę współpracujących osób, a w takiej sytuacji modyfikator `private` pozwoli na swobodne zmienianie składnika bez martwienia się o wpływ tych zmian na inne klasy w pakiecie.

Domyślny dostęp pakietowy daje wystarczający stopień ukrywania — pamiętajmy, że składnik „pakietowy" jest niedostępny dla programisty używającego klasy. Jest to bardzo przyjemne, ponieważ domyślny stopień dostępu jest tym, którego będziemy najczęściej używać (a także tym, który otrzymamy, jeśli zapomnimy o podaniu jakiegokolwiek modyfikatora dostępu). W typowej sytuacji będziemy zatem zastanawiać się nad składnikami, które chcielibyśmy uczynić publicznymi dla programisty-klienta. W rezultacie można początkowo dojść do wniosku, że słowa kluczowego `private` nie będziemy używać często, ponieważ można sobie bez niego poradzić (widać tu wyraźną różnicę w stosunku do C++). Spójne używanie modyfikatora `private` jest jednak bardzo istotne, szczególnie jeśli w grę wchodzi wielowątkowość (przekonamy się o tym w rozdziale „Współbieżność").

Oto przykład użycia słowa kluczowego `private`:

```
//: access/IceCream.java
// Demonstracja działania słowa "private".

class Sundae {
```

```
  private Sundae() {}
  static Sundae makeASundae() {
    return new Sundae();
  }
}

public class IceCream {
  public static void main(String[] args) {
    //! Sundae x = new Sundae();
    Sundae x = Sundae.makeASundae();
  }
} ///:~
```

Widzimy tu, jak pomocny może być modyfikator `private`: chcielibyśmy kontrolować sposób tworzenia obiektu i uniemożliwić bezpośredni dostęp do określonego konstruktora (lub wszystkich konstruktorów). W powyższym przykładzie nie możemy stworzyć obiektu typu Sundae poprzez konstruktor — musimy zamiast tego wywoływać metodę `makeASundae()`, aby stworzyła go za nas[4].

Każdą metodę, co do której jesteśmy pewni, że jest jedynie pomocnicza w danej klasie, możemy uczynić prywatną, aby upewnić się, że nie użyjemy jej przypadkowo w innym miejscu pakietu, uniemożliwiając sobie tym samym jej zmianę lub usunięcie — czyniąc ją prywatną, gwarantujemy sobie zachowanie takich możliwości.

To samo odnosi się do prywatnych pól klasy. Jeżeli nie jesteśmy zmuszeni eksponować wewnętrznej implementacji (co zdarza się znacznie rzadziej, niż można przypuszczać), powinniśmy wszystkie pola uczynić prywatnymi. Jednakże to, że referencja do obiektu jest prywatna wewnątrz klasy, nie oznacza, że jakiś inny obiekt nie może posiadać publicznej referencji do tego samego obiektu (dokładniejsze omówienie tych kwestii znajduje się w suplementach do książki, publikowanych online).

protected: dostęp „na potrzeby" dziedziczenia

By zrozumieć działanie modyfikatora `protected`, musimy wybiec nieco do przodu. Po pierwsze, powinieneś być świadomy, że zrozumienie poniższego fragmentu nie jest konieczne do dalszej pracy z książką, aż do wprowadzenia pojęcia *dziedziczenia* (rozdział „Wielokrotne wykorzystanie klas"). Jednak w celu uzyskania pełnego obrazu zamieszczam tutaj pobieżny opis i przykład użycia modyfikatora `protected`.

Słowo kluczowe `protected` jest związane z koncepcją *dziedziczenia*, polegającą na tworzeniu nowej klasy poprzez dodanie nowych składników do klasy już istniejącej (określanej jako *klasa bazowa*), bez jakichkolwiek modyfikacji tej ostatniej. Możliwa jest również zmiana zachowania istniejących składowych. By umożliwić dziedziczenie po istniejącej klasie, musimy zaznaczyć, że nowa klasa rozszerza tę pierwszą. Robimy to w następujący sposób:

```
class Foo extends Bar {
```

Reszta definicji klasy powinna wyglądać normalnie.

[4] W tym przypadku występuje także inny efekt: konstruktor domyślny jest jedynym zdefiniowanym, a jednocześnie jest on prywatny. W rezultacie niemożliwe będzie dziedziczenie po tej klasie (ten temat zostanie wprowadzony później).

Jeżeli tworzymy nowy pakiet i dziedziczymy po klasie z innego pakietu, wtedy jedynymi składowymi, do których mamy dostęp, są składowe publiczne pakietu oryginalnego (oczywiście jeżeli przeprowadzimy dziedziczenie *w tym samym* pakiecie, będziemy korzystać ze wszystkich składowych dostępnych w ramach pakietu). Czasami twórca klasy bazowej pragnie udzielić dostępu do określonego składnika klasom pochodnym, jednak nie chce go udzielać całemu światu. Do tego właśnie służy modyfikator `protected`.

Odwołamy się teraz jeszcze raz do pliku *Cookie.java*. Następująca klasa *nie posiada* dostępu do składnika „przyjaznego":

```
//: access/ChocolateChip.java
// Brak "pakietowego dostępu" do składnika z innego pakietu.
import access.dessert.*;

public class ChocolateChip extends Cookie {
  public ChocolateChip() {
   System.out.println("Konstruktor ChocolateChip");
  }
  public void chomp() {
    //! bite(); // Brak dostępu do bite
  }
  public static void main(String[] args) {
    ChocolateChip x = new ChocolateChip();
    x.chomp();
  }
} /* Output:
Konstruktor Cookie
Konstruktor ChocolateChip
*///:~
```

Jedną z interesujących spraw dotyczących dziedziczenia jest to, że jeżeli metoda `bite()` istnieje w klasie `Cookie`, to istnieje również w każdej klasie dziedziczącej po `Cookie`. Ponieważ jednak `bite()` jest składową dostępną w ramach pakietu, lecz zdefiniowaną w innym pakiecie, w naszym pakiecie nie możemy z niej korzystać. Moglibyśmy oczywiście uczynić ją publiczną, jednak to dałoby do niej dostęp każdemu, a być może tego nie chcemy. Zmieniając klasę `Cookie` w następujący sposób:

```
//: access/cookie2/Cookie.java
package access.cookie2;

public class Cookie {
  public Cookie() {
    System.out.println("Konstruktor Cookie");
  }
  protected void bite() {
    System.out.println("chrup");
  }
} ///:~
```

spowodujemy, że `bite()` będzie dostępna dla wszystkich klas dziedziczących po `Cookie`:

```
//: access/ChocolateChip2.java
import access.cookie2.*;

public class ChocolateChip2 extends Cookie {
  public ChocolateChip2() {
   System.out.println("Konstruktor ChocolateChip2");
```

```
  }
  public void chomp() { bite(); } // Metoda chroniona
  public static void main(String[] args) {
    ChocolateChip2 x = new ChocolateChip2();
    x.chomp();
  }
} /* Output:
Konstruktor Cookie
Konstruktor ChocolateChip2
chrup
*///:~
```

Zauważ, że choć metoda `bite()` wciąż pozwala na dostęp „pakietowy", *nie jest* jednak publiczna.

Ćwiczenie 4. Wykaż, że metody chronione podlegają dostępowi pakietowemu, ale nie są metodami publicznymi (2).

Ćwiczenie 5. Utwórz klasę ze składowymi i metodami publicznymi, chronionymi i prywatnymi, a także podlegające dostępowi pakietowemu. Utwórz obiekt tej klasy i sprawdź, jakie komunikaty wygeneruje kompilator w reakcji na próby odwołań do wszystkich metod i pól klasy. Pamiętaj, że klasy z tego samego katalogu wchodzą w skład pakietu domyślnego (2).

Ćwiczenie 6. Utwórz klasę z polami chronionymi i drugą (w tym samym pliku), która posiada metodę manipulującą danymi chronionymi z pierwszej klasy (1).

Interfejs i implementacja

Kontrola dostępu jest często nazywana *ukrywaniem implementacji*. Grupowanie danych i metod w klasy w połączeniu z ukrywaniem danych jest często nazywane *hermetyzacją* lub *enkapsulacją* (ang. *encapsulation*)[5]. W rezultacie otrzymujemy typ danych o pewnej charakterystyce i możliwych zachowaniach.

Kontrola dostępu wprowadza rozgraniczenia wewnątrz takiego typu z dwóch istotnych powodów. Pierwszym jest określenie, czego programista-klient powinien używać, a czego nie. Możemy dzięki temu wbudować nasze wewnętrzne mechanizmy w tworzoną strukturę bez przejmowania się tym, że klient przypadkowo potraktuje „wnętrzności" jako część interfejsu, którego ma używać.

Łączy się to bezpośrednio z drugim powodem — chęcią oddzielenia interfejsu od implementacji. Jeżeli nasza struktura jest wykorzystywana w pewnym zbiorze programów, jednak programiści-klienci nie mogą zrobić niczego poza wysyłaniem komunikatów do publicznego interfejsu, wtedy możemy zmienić wszystko, co *nie* jest publiczne (na przykład jest dostępne w ramach pakietu, chronione lub prywatne) bez konieczności zmian w kodzie klientów.

[5] Niektórzy określają tym terminem samo ukrywanie implementacji.

Dla czytelności można przyjąć styl tworzenia klas polegający na umieszczeniu najpierw składowych publicznych, potem chronionych, „przyjaznych", i na końcu prywatnych. Zaletą takiego rozwiązania jest to, że użytkownik klasy może wtedy — czytając od góry do dołu — zobaczyć na początku to, co jest dla niego istotne (składowe publiczne, ponieważ ma do nich dostęp spoza pliku), a następnie przerwać czytanie, gdy napotka składniki niepubliczne będące częścią wewnętrznej implementacji:

```
//: access/OrganizedByAccess.java

public class OrganizedByAccess {
  public void pub1() { /* ... */ }
  public void pub2() { /* ... */ }
  public void pub3() { /* ... */ }
  private void priv1() { /* ... */ }
  private void priv2() { /* ... */ }
  private void priv3() { /* ... */ }
  private int i;
  // ...
} ///:~
```

Metoda ta czyni kod tylko częściowo prostszym w czytaniu, ponieważ interfejs i implementacja są wciąż ze sobą wymieszane, tzn. kod źródłowy — implementacja — znajduje się wewnątrz klasy. Dokumentacja tworzona na podstawie komentarzy, generowana z użyciem narzędzia Javadoc, zmniejsza dodatkowo znaczenie czytelności kodu dla programisty-klienta. Wyświetlanie interfejsu dla użytkownika klasy jest tak naprawdę zadaniem *przeglądarki klas* (ang. *class browser*) — narzędzia potrafiącego przyjrzeć się wszystkim dostępnym klasom i pokazać w użyteczny sposób, co można z nimi zrobić (tj. jakie ich składniki są dostępne). W języku Java przy przeglądaniu dokumentacji JDK rolę przeglądarki klas pełni z powodzeniem najzwyklejsza przeglądarka WWW.

Dostęp do klas

W Javie możliwe jest również użycie modyfikatorów dostępu w celu określenia, które klasy *w* bibliotece będą dostępne dla użytkowników tej biblioteki. Jeżeli chcemy, aby klasa była dostępna dla programisty-klienta, wtedy umieszczamy słowo kluczowe `public` gdzieś przed jej definicją. Konstrukcja taka decyduje, że klient będzie w ogóle w stanie stworzyć obiekt takiej klasy.

Aby udzielić dostępu do klasy, specyfikator musi pojawić się przed słowem kluczowym `class`. Możemy zatem napisać:

```
public class Widget {
```

Teraz, zakładając, że nasza biblioteka nazywa się `access`, każdy klient może zacząć używać klasy `Widget`, pisząc:

```
import access.Widget;
```

lub:

```
import access.*;
```

Nałożono jednak szereg dodatkowych ograniczeń:

1. W jednostce kompilacji (pliku) może znajdować się tylko jedna klasa publiczna. Idea polega na tym, że każda jednostka kompilacji ma jeden publiczny interfejs reprezentowany przez klasę publiczną. Może posiadać dowolnie wiele pomocniczych klas dostępnych w obrębie pakietu. Jeżeli jednak w jednostce kompilacji umieścimy więcej niż jedną klasę publiczną, kompilator zaprotestuje.
2. Nazwa klasy publicznej musi dokładnie pasować do nazwy pliku zawierającego jednostkę kompilacji, łącznie z wielkością liter. Tak więc dla klasy `Widget` plik musi nazywać się *Widget.java*, nie zaś *widget.java* czy *WIDGET.java*. Także w tym przypadku otrzymamy błąd kompilacji, jeśli na to nie przystaniemy.
3. Możliwe, choć nietypowe, jest posiadanie jednostki kompilacji niezawierającej żadnej klasy publicznej. W takim przypadku nazwa pliku może być dowolna (choć zupełna dowolność nazwy może z kolei zmylić przyszłych opiekunów kodu).

Co się dzieje, gdy wewnątrz biblioteki access mamy jakąś klasę wykorzystywaną jedynie do osiągnięcia celu realizowanego przez `Widget` lub inną publiczną klasę z access? Nie chcemy zajmować się przygotowaniem jej dokumentacji dla programisty-klienta, a poza tym wydaje nam się, iż kiedyś być może wszystko kompletnie zmienimy albo w ogóle pozbędziemy się tej klasy, zastępując ją inną. Aby zapewnić sobie taką elastyczność, musimy upewnić się, iż żaden klient nie uzależni się od szczegółów implementacyjnych ukrytych wewnątrz access. W tym celu po prostu nie umieszczamy słówka `public` przed deklaracją klasy (przez co staje się ona widoczna jedynie z wnętrza pakietu).

Ćwiczenie 7. Utwórz bibliotekę złożoną z fragmentów kodu dotyczących access i `Widget`. Utwórz obiekt klasy `Widget` w klasie spoza pakietu access (1).

Tworząc klasę dostępną w obrębie pakietu, wciąż warto niektóre z jej pól uczynić prywatnymi — pola zawsze powinny być jak najbardziej „prywatne"; jeśli natomiast chodzi o metody, to przeważnie rozsądnym rozwiązaniem jest stosowanie tego samego dostępu do nich jak klas, w których się znajdują (czyli dostępu pakietowego). Klasy dostępne w obrębie pakietu są zazwyczaj używane wyłącznie wewnątrz niego, a zatem metody tych klas należy definiować jako publiczne wyłącznie w razie konieczności, a w tych przypadkach kompilator nas o tym poinformuje.

Zauważmy, że klasa nie może być prywatna (uczyniłoby to ją niedostępną dla wszystkich z wyjątkiem jej samej) lub chroniona[6]. Tak więc w kwestii stopnia dostępności klasy istnieją jedynie dwie możliwości: klasa dostępna w obrębie pakietu i klasa „publiczna". Jeżeli nie chcemy, aby ktokolwiek miał dostęp do klasy, wtedy możemy uczynić wszystkie konstruktory prywatnymi, uniemożliwiając tym samym wszystkim, poza nami, utworzenie obiektu naszej klasy wewnątrz statycznej składowej klasy. Oto przykład:

```
//: access/Lunch.java
// Używa specyfikatorów dostępu do klasy, czyniąc klasę
// klasą prywatną, z prywatnymi konstruktorami:
```

[6] Tak naprawdę *klasy wewnętrzne* mogą być prywatne lub chronione, jednak jest to przypadek specjalny. Zostanie to opisane w rozdziale „Klasy wewnętrzne".

```
class Soup1 {
  private Soup1() {}
  // (1) Udostępnienie konstrukcji przy pomocy metody statycznej:
  public static Soup1 makeSoup() {
    return new Soup1();
  }
}

class Soup2 {
  private Soup2() {}
  // (2) Utworzenie obiektu statycznego i zwrócenie referencji
  // na żądanie (wzorzec projektowy "Singleton"):
  private static Soup2 ps1 = new Soup2();
  public static Soup2 access() {
    return ps1;
  }
  public void f() {}
}

// Tylko jedna klasa publiczna w pliku:
public class Lunch {
  void testPrivate() {
    // Nie wolno! Konstruktor prywatny:
    //! Soup1 soup = new Soup1();
  }
  void testStatic() {
    Soup1 soup = Soup1.makeSoup();
  }
  void testSingleton() {
    Soup2.access().f();
  }
} ///:~
```

Do tej pory nasze metody zwracały jedynie void lub typy podstawowe, dlatego definicja:

```
public static Soup1 access() {
    return new Soup1();
  }
```

może się z początku wydawać nieco dziwna. Słowo znajdujące się przed nazwą metody (access) mówi, co metoda zwraca. Do tej pory było to zwykle słowo void oznaczające, że nic nie jest zwracane. Możemy jednak również zwracać referencję do obiektu, co właśnie ma miejsce w powyższym przykładzie. Ta metoda zwraca referencję do obiektu klasy Soup1.

Klasy Soup1 i Soup2 pokazują, jak uniknąć bezpośredniego tworzenia klasy poprzez uczynienie wszystkich jej konstruktorów prywatnymi. Należy pamiętać, że jeżeli nie stworzymy jawnie choćby jednego konstruktora, wtedy zostanie dla nas wyprodukowany konstruktor domyślny (niepobierający argumentów). Dzięki napisaniu konstruktora domyślnego zapobiegamy jego automatycznemu utworzeniu. Czyniąc go prywatnym, uniemożliwiamy wszystkim utworzenie obiektu naszej klasy. Jak w takim razie ktoś może ją wykorzystać? Wcześniejszy przykład pokazuje dwie możliwości. Po pierwsze, obiekt klasy Soup1 jest tworzony wewnątrz statycznej metody tej klasy, a następnie metoda taka zwraca referencję do utworzonego obiektu. Może to być użyteczne, jeżeli chcemy wykonać jakieś dodatkowe operacje na obiekcie przed zwróceniem referencji do niego lub jeśli chcemy zliczać stworzone obiekty typu Soup1 (np. w celu ograniczenia ich populacji).

Klasa Soup2 realizuje jeden z tzw. *wzorców projektowych* (ang. *design patterns*) — opisane są one w książce *Thinking in Patterns (in Java)*, którą można ściągnąć z witryny *www.MindView.net*. Ten szczególny użyty tutaj wzorzec nazywany jest „singletonem" („jedynakiem"), ponieważ polega na umożliwieniu stworzenia tylko jednego obiektu klasy. Obiekt typu Soup2 jest tworzony jako statyczny i prywatny składnik klasy Soup2, a zatem powstaje w ten sposób jeden i tylko jeden egzemplarz. Następnie możemy go otrzymać poprzez wywołanie publicznej metody access().

Jak już wspomniałem, jeżeli nie podamy żadnego modyfikatora dostępu do klasy, przyjmowany jest domyślny dostęp pakietowy. Oznacza to, że obiekt naszej klasy może zostać stworzony przez każdą inną klasę pakietu, jednak nie poza pakietem. (Należy pamiętać, że pliki znajdujące się w tym samym katalogu, nieposiadające jawnego określenia pakietu stają się niejawnie częścią pakietu domyślnego dla tego katalogu). Jeżeli jednak statyczny składnik klasy jest publiczny, wtedy programista-klient może nadal mieć dostęp do tego składnika, mimo że nie może stworzyć obiektu klasy.

Ćwiczenie 8. Wzorując się na przykładzie *Lunch.java*, napisz klasę ConnectionManager, która będzie zarządzać tablicą obiektów Connection (tablica ma mieć stałą liczbę elementów). Programista-klient nie ma mieć możliwości jawnego tworzenia obiektów klasy Connection, polegając jedynie na statycznej metodzie z klasy ConnectionManager. Kiedy klasie ConnectionManager skończą się obiekty Connection, metoda ta powinna zwracać referencję pustą. Przetestuj działanie klas w metodzie main() (4).

Ćwiczenie 9. Utwórz poniższy plik w katalogu *access/local* (zakładam, że w obrębie CLASSPATH) (2):

```
// access/local/PackagedClass.java
package access.local;

class PackagedClass {
  public PackagedClass() {
    System.out.println("Konstrukcja obiektu klasy pakietowej");
  }
}
```

Następnie w katalogu poza *access/local* umieść taki plik:

```
// access/foreign/Foreign.java
package access.foreign;
import access.local.*;

public class Foreign {
  public static void main(String[] args) {
    PackagedClass pc = new PackagedClass();
  }
}
```

Wyjaśnij, dlaczego kompilator zgłasza błąd. Czy uczynienie klasy Foreign częścią pakietu access.local cokolwiek zmieni w zachowaniu kompilatora (2)?

Podsumowanie

W każdej relacji istotna jest obecność rozgraniczeń respektowanych przez wszystkie uczestniczące w niej strony. Tworząc bibliotekę, ustanawiamy relację z jej użytkownikiem — programistą, który z użyciem naszej biblioteki próbuje napisać aplikację lub stworzyć większą bibliotekę.

Gdybyśmy nie ustanowili reguł, klient mógłby zrobić wszystko z każdą składową klasy, nawet jeśli wolelibyśmy, aby nie manipulował bezpośrednio którąś z nich. Wszystko byłoby odsłonięte przed światem.

Rozdział ten pokazał, jak z klas formuje się biblioteki — po pierwsze, w jaki sposób grupa klas jest pakowana w bibliotekę, i po drugie, w jaki sposób klasa kontroluje dostęp do swoich składników.

Ocenia się, że projekt programistyczny prowadzony w języku C zaczyna załamywać się gdzieś pomiędzy 50 a 100 tysiącami wierszy kodu, ponieważ C posiada pojedynczą przestrzeń nazw, a zatem nazwy zaczynają wchodzić w kolizje, powodując dodatkowy narzut związany z zarządzaniem nimi. W Javie słowo kluczowe `package`, schemat nazywania pakietów oraz instrukcja `import` dają nam całkowitą kontrolę nad nazwami, zatem problem ich kolizji nie występuje.

Istnieją dwa powody wprowadzenia kontroli dostępu do składowych klas. Pierwszym jest trzymanie użytkowników z daleka od tego, czego nie powinni dotykać. To elementy niezbędne dla wewnętrznych działań typu danych, nie będących jednak częścią interfejsu, który klienci powinni wykorzystywać w celu rozwiązywania konkretnych problemów. Uczynienie metod i pól prywatnymi służy więc użytkownikom, ponieważ pozwala im na łatwe odróżnienie tego, co jest dla nich istotne, od tego, co mogą zignorować. Ułatwia im zrozumienie klasy.

Drugim z najważniejszych powodów wprowadzenia kontroli dostępu jest umożliwienie projektantowi biblioteki zmiany wewnętrznego sposobu działania klasy bez przejmowania się wpływem, jaki będzie to miało na programistę-klienta. Możemy zbudować najpierw klasę w jakiś sposób, a następnie odkryć, że zmiana struktury kodu przyniesie znacznie większą szybkość. Jeżeli interfejs i implementacja są od siebie wyraźnie oddzielone i chronione, można to osiągnąć bez zmuszania użytkowników do przepisywania ich kodu. Kontrola dostępu gwarantuje, że żaden programista-klient nie uzależni swojego projektu od części kodu stanowiącej implementację klasy.

Dzięki możliwości zmiany wewnętrznej implementacji możemy nie tylko ulepszać nasz projekt w późniejszym czasie, ale i popełniać błędy. Błędy zostaną zresztą popełnione bez względu na to, jak starannie planujemy i projektujemy. Wiedząc, że popełnianie błędów jest stosunkowo bezpieczne, stajemy się bardziej skorzy do eksperymentowania, szybciej się uczymy i szybciej kończymy nasze zadanie.

Publiczny interfejs jest tym, *co widzi* użytkownik klasy, dlatego w jego przypadku najważniejsze jest poprawne zdefiniowanie go już w fazach analizy i projektowania. Nawet tu jednak mamy pewien margines dla zmian. Jeśli zdefiniowany za pierwszym razem interfejs nie jest dobry, możemy bezpiecznie *dodawać* do niego metody, jeżeli tylko nie usuwamy żadnej, która była już wykorzystana w kodzie programistów-klientów.

Zauważ, że kontrola dostępu dotyczy relacji — rodzaju komunikacji — pomiędzy twórcą biblioteki a jej użytkownikami. Bywają jednak i inne przypadki. Na przykład, jeśli piszemy kod dla siebie albo pracujemy w niewielkich zespołach zasiedlających blisko położone biura, możemy umieszczać wszystko w jednym pakiecie. Tu występuje bowiem inny rodzaj komunikacji, przez co wyrażanie reguł kontroli dostępu w kodzie może być zbędne — wystarczy dostęp pakietowy i ustalenia poczynione na spotkaniach.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 7.

Wielokrotne wykorzystanie klas

Jedną z najbardziej użytecznych cech Javy jest możliwość powtórnego wykorzystania kodu. Aby ta cecha zasługiwała na uwagę, musi być czymś więcej niż zwykłym kopiowaniem kodu i dopasowywaniem go do potrzeb.

Takie rozwiązanie stosuje się w językach proceduralnych podobnych do C, lecz nie działa ono zbyt dobrze. Natomiast w Javie rozwiązanie tego problemu oparto, jak wszystko, na klasach. Powtórne użycie kodu jest możliwe dzięki tworzeniu nowych klas, ale zamiast tworzyć je od podstaw, wykorzystuje się klasy już istniejące, które zbudowali i przetestowali inni.

Cała sztuka polega na korzystaniu z istniejących klas bez zmian w ich kodzie. W tym rozdziale przedstawię dwa sposoby osiągnięcia tego celu. Pierwszy jest bardzo prosty: tworzymy obiekty klas już istniejących jako składowe nowych klas. Technika ta nazywa się *kompozycją*, gdyż nowa klasa jest skomponowana z obiektów już istniejących klas. Po prostu wykorzystujemy ponownie funkcje kodu, a nie jego postać.

Drugie rozwiązanie jest bardziej wyrafinowane. Polega na stworzeniu nowej klasy jako *typu* klasy istniejącej. Do klasy istniejącej dodajemy nowy kod bez modyfikacji istniejącego, a całą resztę robi już kompilator. To „magiczne" postępowanie nazywa się *dziedziczeniem*. Dziedziczenie jest jednym z kamieni węgielnych programowania obiektowego i wynikają z niego dodatkowe zagadnienia, które zostały opisane w rozdziale „Polimorfizm".

Okazuje się, że składnia i zachowanie w obu tych rozwiązaniach są podobne (ma to sens, gdyż oba rozwiązania są metodami uzyskiwania nowych typów z już istniejących). W tym rozdziale zapoznasz się właśnie z tymi mechanizmami powtórnego wykorzystania kodu.

Składnia kompozycji

Do tej pory kompozycję stosowaliśmy dosyć często. Wystarczy po prostu umieścić odwołania do obicktów wewnątrz nowo tworzonych klas. Przypuśćmy, że chcielibyśmy utworzyć obiekt przechowujący kilka obiektów klasy `String`, parę zmiennych typu podstawowego i jakiś obiekt innej klasy. Typy podstawowe definiuje się bezpośrednio, a w pozostałych przypadkach używa się referencji:

```
//: reusing/SprinklerSystem.java
// Kompozycja w służbie wielokrotnego wykorzystania kodu.

class WaterSource {
  private String s;
  WaterSource() {
    System.out.println("WaterSource()");
    s = "Skonstruowano";
  }
  public String toString() { return s; }
}

public class SprinklerSystem {
  private String valve1, valve2, valve3, valve4;
  private WaterSource source = new WaterSource();
  private int i;
  private float f;
  public String toString() {
    return
      "valve1 = " + valve1 + " " +
      "valve2 = " + valve2 + " " +
      "valve3 = " + valve3 + " " +
      "valve4 = " + valve4 + "\n" +
      "i = " + i + " " + "f = " + f + " " +
      "source = " + source;
  }
  public static void main(String[] args) {
    SprinklerSystem sprinklers = new SprinklerSystem();
    System.out.println(sprinklers);
  }
} /* Output:
WaterSource()
valve1 = null valve2 = null valve3 = null valve4 = null
i = 0 f = 0.0 source = Skonstruowano
*///:~
```

Jedna z metod zdefiniowanych w obu klasach — o nazwie `toString()` — jest metodą specjalną. W dalszej części książki dowiesz się, że każdy obiekt posiada metodę `toString()`, która jest wywoływana w sytuacjach szczególnych — kiedy kompilator żąda łańcucha tekstowego, a dysponuje jedynie obiektem. Tak więc w poniższym wyrażeniu umieszczonym w metodzie `SprinklerSystem.toString()`:

```
"source = " + source;
```

kompilator widzi, że próbujemy dodać obiekt typu `String` (jakim jest `source` =) do obiektu klasy `WaterSource`. Jest to bezsensowne, ponieważ można „dodawać" wyłącznie `String` do innego obiektu typu `String`, toteż dokładnie oznacza to: „Zamień `source` na łańcuch tekstowy, wywołując metodę `toString()`!". Po wykonaniu tego zadania oba łań-

cuchy tekstowe zostaną połączone, a obiekt wynikowy typu `String` przekazany metodzie `System.out.println()` (albo, równie dobrze, naszym metodom statycznym `print()` i `printnb()`). Jeżeli kiedykolwiek chciałbyś uzyskać takie możliwości we własnej klasie, wystarczy, że dopiszesz do niej metodę `toString()`.

Typy podstawowe, będące polami klasy, są automatycznie inicjalizowane zerem, tak jak to opisano w rozdziale „Wszystko jest obiektem", lecz referencje do obiektów są ustawiane na `null`, co przy próbie wywołania metody na rzecz któregoś z tych obiektów spowoduje zgłoszenie wyjątku (błędu czasu wykonania). Wygodne jest zaś to, że puste referencje można wypisać, nie powodując zgłoszenia wyjątku.

To, że kompilator nie tworzy po prostu domyślnego obiektu dla każdej referencji, ma uzasadnienie, ponieważ narażałoby to w wielu przypadkach na niepotrzebne koszty. Jeżeli chcemy, aby odwołania były zainicjowane, można to zrobić:

1. W miejscu definiowania obiektów. Oznacza to, że zawsze będą inicjowane przed wywołaniem konstruktora.
2. Wewnątrz konstruktora danej klasy.
3. Tuż przed zajściem potrzeby użycia obiektu. Jest to często nazywane *inicjalizacją leniwą*. Może zmniejszyć koszty w przypadkach, gdy tworzenie obiektu jest czasochłonne, a obiekty nie wymagają tworzenia każdorazowo.
4. Za pomocą mechanizmu inicjalizacji egzemplarzy niestatycznych.

Wszystkie cztery sposoby przedstawiono poniżej:

```
//: reusing/Bath.java
// Inicjalizacja wewnątrz konstruktora w przypadku kompozycji.
import static net.mindview.util.Print.*;

class Soap {
  private String s;
  Soap() {
    print("Soap()");
    s = "Skonstruowany";
  }
  public String toString() { return s; }
}

public class Bath {
  private String // Inicjalizacja w miejscu definicji:
    s1 = "Radosny",
    s2 = "Radosny",
    s3, s4;
  private Soap castille;
  private int i;
  private float toy;
  public Bath() {
    print("Wewnątrz Bath()");
    s3 = "Uradowany";
    toy = 3.14f;
    castille = new Soap();
  }
  // Blok inicjalizacji egzemplarza:
  { i = 47; }
```

```
  public String toString() {
    if(s4 == null) // Inicjalizacja leniwa:
      s4 = "Uradowany";
    return
      "s1 = " + s1 + "\n" +
      "s2 = " + s2 + "\n" +
      "s3 = " + s3 + "\n" +
      "s4 = " + s4 + "\n" +
      "i = " + i + "\n" +
      "toy = " + toy + "\n" +
      "castille = " + castille;
  }
  public static void main(String[] args) {
    Bath b = new Bath();
    print(b);
  }
} /* Output:
Wewnątrz Bath()
Soap()
s1 = Radosny
s2 = Radosny
s3 = Uradowany
s4 = Uradowany
i = 47
toy = 3.14
castille = Skonstruowany
*///:~
```

Warto zauważyć, że instrukcja wypisująca wewnątrz konstruktora klasy `Bath` jest wykonywana, zanim nastąpi jakakolwiek inicjalizacja. Jeśli inicjalizacja nie nastąpi w miejscu definicji, to nadal nie mamy gwarancji, że dokona się przed przesłaniem komunikatu do referencji reprezentującej obiekt — nieuchronnie objawi się to wyjątkiem w czasie działania programu.

Metoda `toString()` wypełnia obiekt wskazywany przez `s4`, toteż wszystkie pola są prawidłowo zainicjowane przed wykorzystaniem.

Ćwiczenie 1. Napisz prostą klasę. Wewnątrz drugiej klasy zdefiniuj pole dla obiektu pierwszej klasy i zastosuj leniwą inicjalizację, by stworzyć instancję tego obiektu (2).

Składnia dziedziczenia

Dziedziczenie jest integralną częścią Javy (oraz ogólnie języków obiektowych). Okazuje się, że dziedziczenie jest używane zawsze, kiedy tworzymy jakąś klasę, gdyż jeżeli nawet nie dziedziczymy bezpośrednio z innej klasy, to automatycznie dziedziczymy ze standardowej głównej klasy bazowej o nazwie `Object`.

Składnia kompozycji jest oczywista, ale by zastosować mechanizm dziedziczenia, używamy zupełnie innego zapisu. Kiedy dziedziczymy, mówimy: „Ta nowa klasa będzie podobna do tamtej klasy". Aby stwierdzić to samo w kodzie, należy jak zwykle podać nazwę klasy, ale przed nawiasem otwierającym ciało klasy zamieścić słowo kluczowe `extends` (rozszerza), a za nim nazwę *klasy bazowej*. Kiedy już to zrobimy, automatycznie zyskamy wszystkie zmienne i metody składowe klasy bazowej. Oto przykład:

```
//: reusing/Detergent.java
// Składnia i właściwości dziedziczenia.
import static net.mindview.util.Print.*;

class Cleanser {
  private String s = "Cleanser";
  public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public String toString() { return s; }
  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    print(x);
  }
}

public class Detergent extends Cleanser {
  // Zmiana metody:
  public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Wywołanie wersji z klasy bazowej
  }
  // Uzupełnienie interfejsu o nowe metody:
  public void foam() { append(" foam()"); }
  // Test nowej klasy pochodnej:
  public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    print(x);
    print("Test klasy bazowej:");
    Cleanser.main(args);
  }
} /* Output:
Cleanser dilute() apply() Detergent.scrub() scrub() foam()
Test klasy bazowej:
Cleanser dilute() apply() scrub()
*///:~
```

Przykład ten pokazuje wiele ciekawych możliwości. Po pierwsze, w metodzie `append()` klasy `Cleanser` obiekty `String` są łączone w łańcuch `s` poprzez zastosowanie operatora `+=` będącego jednym z operatorów (wraz z `+`), które zostały *przeciążone* przez projektantów Javy do pracy z łańcuchami tekstowymi.

Po drugie, zarówno `Cleanser`, jak i `Detergent` zawierają metodę `main()`. Można stworzyć metodę `main()` dla każdej z klas; technika umieszczania metody `main()` w każdej z klas pozwala na łatwe testowanie. Nie ma konieczności usuwania metody `main()` po zakończeniu testów, można ją pozostawić do późniejszego sprawdzenia.

Jeśli w programie będzie wiele klas, to zostanie uruchomiona tylko metoda `main()` klasy wywołanej z wiersza poleceń. Tak więc jeżeli wydamy polecenie `java Detergent`, zostanie wywołana metoda `Detergent.main()`. Można również wywołać `java Cleanser`,

aby uruchomić `Cleanser.main()`, nawet jeśli klasa `Cleanser` nie jest poprzedzona słowem `public`. Bo nawet przy domyślnym dostępie pakietowym dostępna jest publiczna metoda `main()` klasy.

W przykładzie widać wywołanie `Cleanser.main()` bezpośrednio z metody `Detergent.main()` z podaniem tych samych argumentów wiersza poleceń (oczywiście można przekazać jej dowolną tablicę łańcuchów tekstowych).

Istotne jest, że wszystkie metody klasy `Cleanser` są publiczne. Pamiętaj, że jeżeli nie określisz żadnego specyfikatora dostępu dla składowej, to domyślnie będzie on „przyjazny", co pozwoli na dostęp jedynie składowym pakietu. Tym sposobem *wewnątrz tego pakietu*, bez podania specyfikatora dostępu, każdy mógłby użyć tych metod. Klasa `Detergent` nie powinna mieć zatem z tym żadnych kłopotów. Jednak gdyby jakaś klasa z innego pakietu dziedziczyła po `Cleanser`, to mogłaby jedynie sięgnąć do jej składowych publicznych. Aby umożliwić dziedziczenie, stosuj zasadę określania wszystkich pól jako `private`, a metod jako `public` (na dostęp z klas pochodnych zezwalają również składowe typu `protected` — pokażę to w dalszej części). Oczywiście w konkretnych przypadkach powinieneś się dostosować do potrzeb, ale niech to będzie pożyteczna wskazówka.

Klasa `Cleanser` zawiera w swoim interfejsie zestaw metod: `append()`, `dilute()`, `apply()`, `scrub()` oraz `print()`. Ponieważ klasa `Detergent` jest *wywiedziona* z `Cleanser` (poprzez słowo kluczowe `extends`), to automatycznie zyskuje wszystkie te metody w swoim interfejsie — mimo że nie widać ich bezpośredniej definicji. Można sobie wyobrazić dziedziczenie jako *powtórne wykorzystanie klasy*.

Jak widać na przykładzie metody `scrub()`, możliwa jest modyfikacja metody klasy bazowej. W takim przypadku może się przydać wywołanie metody pierwotnej wewnątrz tej nowej. Wewnątrz `scrub()` nie można jednak zwyczajnie wywołać `scrub()`, gdyż spowoduje to powstanie wywołania rekurencyjnego, czego akurat tutaj nie chcemy. Możliwość wywołania metody pierwotnej daje w Javie słowo kluczowe `super`, które odwołuje się do „nadklasy" — klasy, po której dziedziczy klasa bieżąca. Tak więc wyrażenie `super.scrub()` wywoła metodę `scrub()`z klasy bazowej.

Stosując dziedziczenie, wcale nie jesteśmy ograniczeni do wykorzystywania metod klasy bazowej. Można również dodawać nowe metody do klasy pochodnej, dokładnie w ten sam sposób, w jaki dodaje się dowolną metodę do klasy — zwyczajnie ją definiując. Metoda o nazwie `foam()` jest tego przykładem.

W ciele metody `Detergent.main()` widać, że na rzecz obiektu `Detergent` można wywołać wszystkie metody dostępne w klasie `Cleanser`, podobnie jak te dodane w `Detergent` (tj. `foam()`).

Ćwiczenie 2. Wyprowadź nową klasę pochodną z klasy `Detergent`. Przesłoń metodę `scrub()` i dodaj nową o nazwie `sterilize()` (2).

Inicjalizacja klasy bazowej

Mamy teraz zamiast jednej dwie klasy powiązane — bazową i jej pochodną, toteż może nie być łatwe wyobrażenie sobie wynikowego obiektu powstałego z klasy pochodnej.

Na zewnątrz wygląda to tak, jakby nowa klasa posiadała taki sam interfejs, jak klasa bazowa, i kilka dodatkowych pól i metod. Jednak dziedziczenie to nie tylko skopiowanie klasy bazowej. Kiedy tworzymy obiekt klasy pochodnej, zawiera on w sobie klasę bazową jako *podobiekt*. Taki podobiekt to jakby osobno utworzony obiekt klasy bazowej. Tak to widać z zewnątrz — podobiekt jest opakowany przez obiekt klasy pochodnej.

Oczywiście istotne jest, by podobiekt klasy bazowej został zainicjowany poprawnie i jest tylko jeden sposób, aby to zapewnić: wykonać inicjalizację w konstruktorze poprzez wywołanie konstruktora klasy bazowej, który dysponuje pełną wiedzą i uprawnieniami, by dokonać takiej inicjalizacji. Wywołania konstruktora klasy bazowej w Javie są dołączane automatycznie w konstruktorach klasy pochodnej. Poniższy przykład pokazuje to w działaniu, w przypadku dziedziczenia trójpoziomowego:

```
//: reusing/Cartoon.java
// Wywołania konstruktora przy dziedziczeniu.
import static net.mindview.util.Print.*;

class Art {
  Art() { print("Konstruktor klasy Art"); }
}

class Drawing extends Art {
  Drawing() { print("Konstruktor klasy Drawing"); }
}

public class Cartoon extends Drawing {
  public Cartoon() { print("Konstruktor klasy Cartoon"); }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
} /* Output:
Konstruktor klasy Art
Konstruktor klasy Drawing
Konstruktor klasy Cartoon
*///:~
```

Widać, że konstruowanie przebiega „od góry", tak więc klasa bazowa jest inicjowana, zanim konstruktor klasy pochodnej może uzyskać do niej dostęp. Nawet jeśli nie napiszemy konstruktora dla klasy `Cartoon()`, to kompilator dołoży za nas konstruktor domyślny, który będzie wywoływał konstruktor klasy bazowej.

Ćwiczenie 3. Udowodnij powyższe stwierdzenie (2).

Ćwiczenie 4. Udowodnij, że konstruktory klasy bazowej są: (a) wywoływane zawsze i (b) wywoływane przed konstruktorami klasy pochodnej (2).

Ćwiczenie 5. Stwórz dwie klasy `A` i `B` z domyślnymi konstruktorami (z pustą lista argumentów), które siebie przedstawiają. Wywiedź nową klasę o nazwie `C` z klasy `A` i stwórz składową klasy `B` wewnątrz `C`. Nie twórz konstruktora klasy `C`. Stwórz obiekt klasy `C` i zaobserwuj, jak to działa (1).

Konstruktory z argumentami

Powyższy przykład zawiera konstruktory domyślne, tj. niepobierające żadnych parametrów. Ich wywołanie przez kompilator jest proste, gdyż nie powstaje pytanie, jakie parametry przekazać. Jeśli klasa nie zawiera parametrów domyślnych lub gdy chcemy wywołać sparametryzowany konstruktor klasy bazowej, trzeba zastosować słowo super i podać odpowiednie argumenty:

```
//. reusing/Chess.java
// Dziedziczenie, konstruktory i argumenty.
import static net.mindview.util.Print.*;

class Game {
  Game(int i) {
    print("Konstruktor klasy Game");
  }
}

class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    print("Konstruktor klasy BoardGame");
  }
}

public class Chess extends BoardGame {
  Chess() {
    super(11);
    print("Konstruktor klasy Chess");
  }
  public static void main(String[] args) {
    Chess x = new Chess();
  }
} /* Output:
Konstruktor klasy Game
Konstruktor klasy BoardGame
Konstruktor klasy Chess
*///:~
```

Jeżeli nie wywołamy konstruktora klasy bazowej w `BoardGame()`, kompilator będzie informował, że nie może odnaleźć konstruktora postaci `Game()`. W dodatku wywołanie konstruktora klasy bazowej *musi* być pierwszą instrukcją w konstruktorze klasy pochodnej (w innym przypadku kompilator przypomni nam o tym).

Ćwiczenie 6. Za pomocą pliku *Chess.java* udowodnij stwierdzenie z poprzedniego akapitu (1).

Ćwiczenie 7. Zmodyfikuj ćwiczenie 5. tak, aby klasy `A` i `B` posiadały konstruktory z parametrami zamiast domyślnych. Dopisz konstruktor dla klasy `C` i wykonaj całą inicjalizację wewnątrz konstruktora klasy `C` (1).

Ćwiczenie 8. Utwórz klasę bazową z jedynym konstruktorem innym niż konstruktor domyślny oraz klasę pochodną z konstruktorem domyślnym (bezargumentowym) i konstruktorem niedomyślnym. W konstruktorach klasy pochodnej wywołaj konstruktor klasy bazowej (1).

Ćwiczenie 9. Napisz klasę o nazwie Root zawierającą egzemplarz każdej z klas (które również napiszesz) o nazwach: Component1, Component2 i Component3. Wyprowadź z klasy Root klasę pochodną Stem, która również będzie zawierała każdy z „komponentów". Wszystkie klasy powinny mieć konstruktory domyślne wypisujące komunikat o klasie (2).

Ćwiczenie 10. Zmodyfikuj poprzednie ćwiczenie tak, by każda klasa miała wyłącznie konstruktor niebędący domyślnym (1).

Delegacje

Trzeci rodzaj relacji, nieobsługiwany wprost przez język Java, nosi nazwę *delegacji*. To relacja pośrednia pomiędzy dziedziczeniem a kompozycją, ponieważ zakłada osadzanie w budowanej klasie obiektu innej klasy, ale z równoczesną ekspozycją wszystkich metod tego obiektu w nowej klasie (jak w dziedziczeniu). Za przykład posłuży statek kosmiczny, który potrzebuje modułu sterującego:

```
//: reusing/SpaceShipControls.java

public class SpaceShipControls {
  void up(int velocity) {}
  void down(int velocity) {}
  void left(int velocity) {}
  void right(int velocity) {}
  void forward(int velocity) {}
  void back(int velocity) {}
  void turboBoost() {}
} ///:~
```

Właściwy statek kosmiczny możemy zmontować za pomocą dziedziczenia:

```
//: reusing/SpaceShip.java

public class SpaceShip extends SpaceShipControls {
  private String name;
  public SpaceShip(String name) { this.name = name; }
  public String toString() { return name; }
  public static void main(String[] args) {
    SpaceShip protector = new SpaceShip("ORP Zwiadowca");
    protector.forward(100);
  }
} ///:~
```

Ale SpaceShip nie pozostaje tak naprawdę w relacji „jest typu" z klasą SpaceShipControls, mimo że można nakazać obiektowi SpaceShip wykonanie forward(). Model byłby dokładniejszy, gdybyśmy powiedzieli, że SpaceShip zawiera w sobie SpaceShipControls, a wszystkie metody SpaceShipControls są eksponowane w klasie SpaceShip. Delegacja służy właśnie do tego:

```
//: reusing/SpaceShipDelegation.java

public class SpaceShipDelegation {
  private String name;
  private SpaceShipControls controls =
    new SpaceShipControls();
```

```
  public SpaceShipDelegation(String name) {
    this.name = name;
  }
  // Metody delegowane:
  public void back(int velocity) {
    controls.back(velocity);
  }
  public void down(int velocity) {
    controls.down(velocity);
  }
  public void forward(int velocity) {
    controls.forward(velocity);
  }
  public void left(int velocity) {
    controls.left(velocity);
  }
  public void right(int velocity) {
    controls.right(velocity);
  }
  public void turboBoost() {
    controls.turboBoost();
  }
  public void up(int velocity) {
    controls.up(velocity);
  }
  public static void main(String[] args) {
    SpaceShipDelegation protector =
      new SpaceShipDelegation("ORP Zwiadowca");
    protector.forward(100);
  }
} ///:~
```

Widać, że wywołania są przekazywane do zawieranego obiektu `controls`; interfejs obiektu zawierającego jest identyczny jak w przypadku dziedziczenia, ale delegacja daje większą kontrolę, bo pozwala choćby na wybiórcze udostępnianie fragmentów interfejsu obiektu zawieranego.

Choć język Java nie obsługuje delegacji wprost, czynią to niektóre narzędzia programistyczne. Powyższy przykład został wygenerowany automatycznie za pośrednictwem środowiska programistycznego JetBrains Java.

Ćwiczenie 11. Zmodyfikuj plik *Detergent.java* tak, aby zastosować w nim delegację (3).

Łączenie kompozycji i dziedziczenia

Stosowanie kompozycji oraz dziedziczenia razem jest dosyć powszechne. Kolejny przykład przedstawia tworzenie bardziej skomplikowanej klasy z zastosowaniem mechanizmów dziedziczenia i kompozycji wraz z konieczną inicjalizacją w konstruktorze:

```
//: reusing/PlaceSetting.java
// Łączenie kompozycji i dziedziczenia.
import static net.mindview.util.Print.*;
```

```
class Plate {
  Plate(int i) {
    print("Konstruktor klasy Plate");
  }
}

class DinnerPlate extends Plate {
  DinnerPlate(int i) {
    super(i);
    print("Konstruktor klasy DinnerPlate");
  }
}

class Utensil {
  Utensil(int i) {
    print("Konstruktor klasy Utensil");
  }
}

class Spoon extends Utensil {
  Spoon(int i) {
    super(i);
    print("Konstruktor klasy Spoon");
  }
}

class Fork extends Utensil {
  Fork(int i) {
    super(i);
    print("Konstruktor klasy Fork");
  }
}

class Knife extends Utensil {
  Knife(int i) {
    super(i);
    print("Konstruktor klasy Knife");
  }
}

// Kulturalny sposób na wykonanie czegoś:
class Custom {
  Custom(int i) {
    print("Konstruktor klasy Custom");
  }
}

public class PlaceSetting extends Custom {
  private Spoon sp;
  private Fork frk;
  private Knife kn;
  private DinnerPlate pl;
  public PlaceSetting(int i) {
    super(i + 1);
    sp = new Spoon(i + 2);
    frk = new Fork(i + 3);
    kn = new Knife(i + 4);
```

```
    pl = new DinnerPlate(i + 5);
    print("Konstruktor klasy PlaceSetting");
  }
  public static void main(String[] args) {
    PlaceSetting x = new PlaceSetting(9);
  }
} /* Output:
Konstruktor klasy Custom
Konstruktor klasy Utensil
Konstruktor klasy Spoon
Konstruktor klasy Utensil
Konstruktor klasy Fork
Konstruktor klasy Utensil
Konstruktor klasy Knife
Konstruktor klasy Plate
Konstruktor klasy DinnerPlate
Konstruktor klasy PlaceSetting
*///:~
```

Choć kompilator wymusi na nas inicjalizację klas bazowych i zamieszczenie jej na początku konstruktora, nie sprawdzi, czy zostały zainicjowane obiekty składowe, toteż trzeba pamiętać, aby zawsze zwracać na to uwagę.

Elegancja rozdziału klas jest zadziwiająca. Do ponownego wykorzystania kodu nie potrzebujemy nawet kodu źródłowego metod — wystarczy zaimportować pakiet (dotyczy to zarówno dziedziczenia, jak i kompozycji).

Zapewnienie poprawnego sprzątania

W Javie nie istnieje pochodzące z C++ pojęcie *destruktora*, czyli metody wywoływanej automatycznie podczas niszczenia obiektu. Przyczyną jest prawdopodobnie to, że w Javie po prostu należy zapomnieć o niszczeniu obiektów i pozostawiać je, pozwalając w zamian odśmiecaczowi pamięci na odzyskanie pamięci w razie potrzeby.

Zazwyczaj jest to dobre rozwiązanie, ale może się zdarzyć, że klasa wykonuje pewne działania podczas swojego życia, które na koniec wymagają wykonania porządków. Jak wspominałem w rozdziale „Inicjalizacja i sprzątanie", nie wiadomo, kiedy zostanie uruchomiony odśmiecacz, a nawet czy w ogóle zostanie uruchomiony. Dlatego jeżeli trzeba po czymś sprzątnąć, należy dopisać specjalną metodę, która to zrobi, i upewnić się, że programiści używający klasy wiedzą, że muszą ją wywołać. Na dodatek — jak piszę w rozdziale „Obsługa błędów za pomocą wyjątków" — należy chronić się przed wyjątkami poprzez zamieszczenie takiego sprzątania w klauzuli `finally`.

Rozważmy przykład systemu projektowania wspomaganego komputerowo, który rysuje obrazki na ekranie:

```
//: reusing/CADSystem.java
// Wymuszanie poprawnego sprzątania.
package reusing;
import static net.mindview.util.Print.*;

class Shape {
  Shape(int i) { print("Konstruktor figury"); }
  void dispose() { print("Usuwanie figury"); }
}
```

```
class Circle extends Shape {
  Circle(int i) {
    super(i);
    print("Rysowanie okręgu");
  }
  void dispose() {
    print("Wymazywanie okręgu");
    super.dispose();
  }
}

class Triangle extends Shape {
  Triangle(int i) {
    super(i);
    print("Rysowanie trójkąta");
  }
  void dispose() {
    print("Wymazywanie trójkąta");
    super.dispose();
  }
}

class Line extends Shape {
  private int start, end;
  Line(int start, int end) {
    super(start);
    this.start = start;
    this.end = end;
    print("Rysowanie linii: " + start + ", " + end);
  }
  void dispose() {
    print("Wymazywanie linii: " + start + ", " + end);
    super.dispose();
  }
}

public class CADSystem extends Shape {
  private Circle c;
  private Triangle t;
  private Line[] lines = new Line[3];
  public CADSystem(int i) {
    super(i + 1);
    for(int j = 0; j < lines.length; j++)
      lines[j] = new Line(j, j*j);
    c = new Circle(1);
    t = new Triangle(1);
    print("Konstruktor łączony");
  }
  public void dispose() {
    print("CADSystem.dispose()");
    // Kolejność sprzątania jest odwrócona
    // względem kolejności inicjalizacji:
    t.dispose();
    c.dispose();
    for(int i = lines.length - 1; i >= 0; i--)
      lines[i].dispose();
    super.dispose();
  }
```

```
    public static void main(String[] args) {
      CADSystem x = new CADSystem(47);
      try {
        // Kod i obsługa wyjątków
      } finally {
        x.dispose();
      }
    }
} /* Output:
Konstruktor figury
Konstruktor figury
Rysowanie linii: 0, 0
Konstruktor figury
Rysowanie linii: 1, 1
Konstruktor figury
Rysowanie linii: 2, 4
Konstruktor figury
Rysowanie okręgu
Konstruktor figury
Rysowanie trójkąta
Konstruktor łączony
CADSystem.dispose()
Wymazywanie trójkąta
Usuwanie figury
Wymazywanie okręgu
Usuwanie figury
Wymazywanie linii: 2, 4
Usuwanie figury
Wymazywanie linii: 1, 1
Usuwanie figury
Wymazywanie linii: 0, 0
Usuwanie figury
Usuwanie figury
*///:~
```

Wszystko w naszym systemie jest figurą — obiektem rodzaju `Shape` (który z kolei jest typu `Object`, ponieważ jest pośrednio wywiedziony z głównej klasy bazowej). Każda z klas przedefiniowuje metodę `dispose()` klasy `Shape` i oprócz tego wywołuje wersję tej metody z klasy bazowej poprzez użycie słowa kluczowego `super`. Wszystkie specjalizowane klasy typu `Shape` — `Circle`, `Triangle` i `Line` — posiadają konstruktor „rysujący", choć dowolna metoda wywołana w czasie życia obiektu może wykonywać coś, co wymaga sprzątnięcia. Każda z klas posiada własną metodę `dispose()` pozwalającą przywrócić rzeczy niezwiązane z pamięcią do stanu przed powstaniem obiektu.

W metodzie `main()` pojawiły się dwa nowe słowa kluczowe, które zostaną dokładnie omówione w rozdziale „Obsługa błędów za pomocą wyjątków": `try` i `finally`. Słowo `try` sygnalizuje, że rozpoczynający się blok (ograniczony nawiasami klamrowymi) jest *obszarem chronionym*, co znaczy, że będzie specjalnie traktowany. Jednym z elementów takiej specjalnej obsługi jest to, że kod zamieszczony w klauzuli `finally`, następującej za obszarem chronionym, będzie wykonywany *zawsze*, bez względu na to, jak zakończy się sam blok `try` (dzięki obsłudze wyjątków możliwe jest opuszczenie bloku `try` na wiele różnych sposobów). W naszym przypadku klauzula `finally` oznacza: „Zawsze wywołaj `dispose()` dla `x`, bez względu na to, co się stanie".

W metodzie sprzątającej (tutaj dispose()) należy także zwrócić uwagę na kolejność wywoływania metod sprzątających dla klasy bazowej i obiektów składowych na wypadek, gdyby jeden podobiekt zależał od innego. Generalnie powinno się stosować taką samą regułę, jaka jest narzucona dla destruktorów przez kompilator C++: najpierw wykonać sprzątanie specyficzne dla danej klasy, w kolejności odwrotnej od tworzenia (zasadniczo wymaga to dostępności elementów klasy bazowej), a następnie, jak to zostało tutaj zaprezentowane, wywołać metodę sprzątającą klasy bazowej.

Istnieje wiele sytuacji, w których kwestia odzyskania pamięci nie będzie żadnym problemem; po prostu pozwalamy odśmiecaczowi pamięci wykonać pracę. Jednak kiedy trzeba zrobić to bezpośrednio, konieczne jest zachowanie szczególnej ostrożności, gdyż jeśli chodzi o odśmiecacz pamięci niewielu rzeczy można być pewnym. Odśmiecanie może przecież w ogóle nie zostać wywołane. Może też odzyskiwać obiekty w takim porządku, jaki sobie wybierze. Najlepiej polegać na odśmiecaczu tylko w przypadku zwrotu pamięci. Jeżeli chcemy, aby miało miejsce jakieś sprzątanie, wpiszmy własne metody i nie polegajmy do końca na metodzie `finalize()`.

Ćwiczenie 12. Do wszystkich klas z ćwiczenia 9. dodaj stosowną hierarchię metod `dispose()` (3).

Ukrywanie nazw

Jeżeli klasa bazowa Javy zawiera metodę, której nazwa jest przeciążona kilka razy, to definicja metody o takiej samej nazwie w klasie pochodnej *nie* spowoduje zakrycia żadnej z wersji zamieszczonych w klasie bazowej (zupełnie inaczej dzieje się w C++). Tym sposobem przeciążanie działa niezależnie od tego, czy metoda była zdefiniowana na tym poziomie czy też w klasie bazowej:

```
//: reusing/Hide.java
// Przeciążanie nazwy metody klasy bazowej w klasie
// pochodnej nie ukrywa wersji z klasy bazowej.
import static net.mindview.util.Print.*;

class Homer {
  char doh(char c) {
    print("doh(char)");
    return 'd';
  }
  float doh(float f) {
    print("doh(float)");
    return 1.0f;
  }
}

class Milhouse {}

class Bart extends Homer {
  void doh(Milhouse m) {
    print("doh(Milhouse)");
  }
}

public class Hide {
```

```
  public static void main(String[] args) {
    Bart b = new Bart();
    b.doh(1);
    b.doh('x');
    b.doh(1.0f);
    b.doh(new Milhouse());
  }
} /* Output:
doh(float)
doh(char)
doh(float)
doh(Milhouse)
*///:~
```

Powyższy przykład pokazuje, że wszystkie przeciążone metody klasy `Homer` są dostępne w klasie `Bart`, pomimo tego, że w klasie tej została zdefiniowana nowa metoda przeciążona (w C++ zdefiniowanie tej metody spowodowałoby ukrycie metod klasy bazowej). Jak zobaczysz w następnym rozdziale, znacznie częściej stosowane jest przesłanianie metod o tej samej nazwie, z dokładnie identyczną sygnaturą i typem zwracanym jak w klasie bazowej. W innym przypadku definicja metody o tej samej nazwie w klasie pochodnej może być myląca (dlatego C++ nie zezwala na to, chroniąc przed prawdopodobnymi błędami).

Wydanie SE5 zostało uzupełnione o adnotację `@Override`, niebędącą słowem kluczowym, ale wykorzystywaną tak jak słowa kluczowe. Kiedy chcemy przesłonić metodę, możemy zastosować taką adnotację; wtedy kompilator tam, gdzie wykryje, że zamiast przesłaniania mamy przeciążanie, wystosuje komunikat o błędzie.

```
//: reusing/Lisa.java
// {CompileTimeError} (Nie skompiluje się)

class Lisa extends Homer {
  @Override void doh(Milhouse m) {
    System.out.println("doh(Milhouse)");
  }
} ///:~
```

Znacznik `{CompileTimeError}` wyłącza powyższy kod z automatycznej, zbiorowej kompilacji przykładów programem Ant; ale kompilacja ręczna spowodowałaby taki komunikat:

```
method does not override a method from its superclass
```

Zapis `@Override` chroni więc przed przypadkowym przeciążeniem tam, gdzie chodzi nam o przesłonięcie.

Ćwiczenie 13. Utwórz klasę z trzykrotnie przeciążoną metodą. Wyprowadź z niej (przez dziedziczenie) nową klasę, z jeszcze jednym przeciążeniem metody, i pokaż, że w klasie pochodnej dostępne są wszystkie cztery wersje metody (2).

Wybór między kompozycją a dziedziczeniem

Oba mechanizmy pozwalają na zamieszczenie podobiektów wewnątrz nowo tworzonej klasy (w przypadku kompozycji jest to jawne, a podczas dziedziczenia — niejawne). Można się zastanawiać, jakie są różnice pomiędzy nimi i który mechanizm w jakiej sytuacji wybrać.

Kompozycja jest zazwyczaj stosowana wtedy, gdy potrzebne są właściwości istniejącej klasy wewnątrz własnej, lecz nie jej interfejs. Osadza się zatem obiekt tak, by można go było wykorzystać w tworzonej klasie, ale użytkownik klasy widzi tylko interfejs nowej klasy bez interfejsu obiektu osadzonego. Aby uzyskać taki efekt, wystarczy zamieścić obiekty klas istniejących wewnątrz nowej klasy jako składowe typu `private`.

Czasami uzasadnione jest zezwolenie użytkownikowi klasy na bezpośredni dostęp do obiektu w niej zamieszczonego — poprzez uczynienie składowych publicznymi. Obiekty składowe same stosują ukrywanie własnej implementacji, a więc jest to nawet bezpieczne. Kiedy użytkownik wie, że zgromadziliśmy grupę części, to interfejs klasy staje się łatwiejszy do zrozumienia. Obiekt `Car`, reprezentujący samochód, będzie tu dobrym przykładem:

```
//: reusing/Car.java
// Kompozycja z obiektami publicznymi.

class Engine {
  public void start() {}
  public void rev() {}
  public void stop() {}
}

class Wheel {
  public void inflate(int psi) {}
}

class Window {
  public void rollup() {}
  public void rolldown() {}
}

class Door {
  public Window window = new Window();
  public void open() {}
  public void close() {}
}

public class Car {
  public Engine engine = new Engine();
  public Wheel[] wheel = new Wheel[4];
  public Door
    left = new Door(),
    right = new Door(); // 2-drzwiowy
  public Car() {
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
```

```
  }
  public static void main(String[] args) {
    Car car = new Car();
    car.left.window.rollup();
    car.wheel[0].inflate(72);
  }
} ///:~
```

Ponieważ sposób, w jaki zostanie dokonana kompozycja samochodu, jest częścią analizy problemu (a nie wewnętrzną sprawą projektu), to utworzenie składowych jako `public` pomaga użytkownikowi zrozumieć, jak stosować klasę, a od twórcy wymaga mniejszej złożoności kodu. Jednak należy mieć na uwadze, że jest to przypadek wyjątkowy i zazwyczaj powinno się tworzyć pola prywatne.

Stosując dziedziczenie, wykorzystuje się istniejącą klasę i tworzy jej specjalizację. Oznacza to, że bierze się klasę ogólnego przeznaczenia i dodaje specjalizację według określonych potrzeb. Po chwili namysłu zauważysz, że bezsensowne byłoby złożenie samochodu z obiektu pojazd — samochód nie zawiera pojazdu, on nim *jest*. Ten właśnie stan — *jest* — zostaje wyrażony przez dziedziczenie, a relacja *ma* poprzez kompozycję.

Ćwiczenie 14. W pliku *Car.java* dodaj metodę `service()` do klasy `Engine` i wywołaj ją w metodzie `main()` (1).

protected

Po zapoznaniu się z dziedziczeniem pora poznać znaczenie słowa kluczowego `protected`. W idealnym świecie składowe prywatne powinny zawsze być ściśle prywatne, ale w rzeczywistych projektach czasem chcemy pozostawić pewne składowe niedostępne publicznie, ale zezwolić na dostęp do nich z metod klas pochodnych.

Słowo kluczowe `protected` służy właśnie do tego. Mówi nam: „To jest prywatne, jeżeli użytkownik jest obcy, ale dostępne dla wszystkiego, co dziedziczy z klasy lub pochodzi z tego samego pakietu". (Tak więc w Javie składowa chroniona jest dostępna w obrębie tego samego pakietu.)

Choć Java pozwala na tworzenie pól (składowych klas) chronionych, najlepszym wyjściem jest pozostawienie danych składowych prywatnymi — zawsze powinno się zachować prawo do zmiany implementacji wewnętrznej. Można zatem pozwolić na kontrolowany dostęp do zmiennej składowej z klas pochodnych poprzez metody chronione:

```
//: reusing/Orc.java
// Słowo "protected".
import static net.mindview.util.Print.*;

class Villain {
  private String name;
  protected void set(String nm) { name = nm; }
  public Villain(String name) { this.name = name; }
  public String toString() {
    return "Jestem łobuzem i mam na imię " + name;
  }
```

```
}

public class Orc extends Villain {
  private int orcNumber;
  public Orc(String name, int orcNumber) {
    super(name);
    this.orcNumber = orcNumber;
  }
  public void change(String name, int orcNumber) {
    set(name); // Dostępna, ponieważ chroniona (nieprywatna)
    this.orcNumber = orcNumber;
  }
  public String toString() {
    return "Ork " + orcNumber + ": " + super.toString();
  }
  public static void main(String[] args) {
    Orc orc = new Orc("Limburger", 12);
    print(orc);
    orc.change("Bob", 19);
    print(orc);
  }
} /* Output:
Ork 12: Jestem łobuzem i mam na imię Limburger
Ork 19: Jestem łobuzem i mam na imię Bob
*///:~
```

Widać, że metoda `change()` ma dostęp do metody `set()`, gdyż ta druga jest chroniona. Warto także zwrócić uwagę na sposób zdefiniowania metody `toString()` klasy `Orc`, która wykorzystuje analogiczną metodę klasy bazowej.

Ćwiczenie 15. Stwórz klasę wewnątrz pakietu, zawierającą metodę chronioną. Spróbuj wywołać tę metodę spoza pakietu i zinterpretuj wynik. Następnie wydziedzicz nową klasę z tej klasy i wywołaj jej metodę chronioną z wnętrza metody klasy pochodnej (2).

Rzutowanie w górę

Najważniejszym aspektem dziedziczenia nie jest to, że zapewnia ono metody naszej nowej klasie. Cechą tą jest związek zachodzący pomiędzy klasą pochodną i bazową. Można go podsumować określeniem, że „nowa klasa *jest typu* klasy istniejącej".

Opis ten nie jest po prostu wymyślnym sposobem tłumaczenia mechanizmu dziedziczenia — jest bowiem bezpośrednio obsługiwany przez język. Jako przykład rozważmy klasę bazową o nazwie `Instrument`, która reprezentuje instrument muzyczny, i klasę z niej wywiedzioną o nazwie `Wind`. Ponieważ dziedziczenie gwarantuje, że wszystkie metody klasy bazowej są również dostępne w klasie pochodnej, to dowolny komunikat, który można wysłać do klasy nadrzędnej, może być przesłany do pochodnej. Jeżeli klasa `Instrument` zawiera metodę `play()`, to ma ją również `Wind`. Możemy zatem powiedzieć, że obiekt klasy `Wind` jest także typu `Instrument`. Poniższy przykład pokazuje, na co w związku z tym pozwala nam kompilator:

```
//: reusing/Wind.java
// Dziedziczenie a rzutowanie w górę.
```

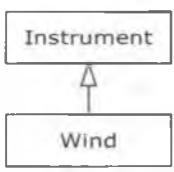
```
class Instrument {
  public void play() {}
  static void tune(Instrument i) {
    // ...
    i.play();
  }
}

// Obiekty Wind są równocześnie typu Instrument,
// bo mają identyczny interfejs:
public class Wind extends Instrument {
  public static void main(String[] args) {
    Wind flute = new Wind();
    Instrument.tune(flute); // Rzutowanie w górę
  }
} ///:~
```

Ciekawa w tym przykładzie jest metoda `tune()`, która pobiera referencję do typu `Instrument`. Wewnątrz metody `Wind.main()` metoda `tune()` jest wywoływana z referencją do obiektu typu `Wind`. Jeśli weźmiemy pod uwagę fakt, że Java jest szczególnie wrażliwa, gdy chodzi o kontrolę typu, może wydać się dziwne, że metoda przyjmująca jeden typ nagle akceptuje inny. Musimy uświadomić sobie, że obiekt klasy `Wind` jest równocześnie obiektem klasy `Instrument` oraz że nie istnieje metoda, którą `tune()` mogłaby wywołać dla klasy `Instrument` nieistniejąca jednocześnie w klasie `Wind`. Kod wewnątrz `tune()` funkcjonuje poprawnie dla obiektów typu `Instrument` oraz jakichkolwiek wywiedzionych z `Instrument`, a czynność konwersji referencji do obiektu klasy `Wind` na referencję do obiektu `Instrument` nazywamy *rzutowaniem w górę* (ang. *upcasting*).

Dlaczego „w górę"

Przyczyna przyjęcia takiego określenia jest historyczna i wywodzi się ze sposobu, w jaki tradycyjnie rysuje się diagramy dziedziczenia — z korzeniem na górze i rozrastający się w dół (oczywiście możesz rysować własne diagramy w sposób, który uważasz za przydatny). Diagram dziedziczenia dla *Wind.java* ma zatem postać:



Rzutowanie od potomka do przodka przesuwa nas w górę diagramu dziedziczenia, toteż jest powszechnie znane jako *rzutowanie w górę*. Rzutowanie to jest zawsze bezpieczne, ponieważ przechodzimy od typu bardziej specyficznego do bardziej ogólnego. Znaczy to, że klasa pochodna jest nadzbiorem klasy bazowej. Może ona zawierać więcej metod niż klasa bazowa, ale musi mieć *przynajmniej* te metody, które są w klasie bazowej. Jedyne, co może się zdarzyć z interfejsem klasy podczas rzutowania w górę, to utrata metod, ale nie jego rozszerzenie. Oto dlaczego kompilator zezwala na rzutowanie w górę bez konieczności wyraźnego określenia rzutowania czy innego specjalnego zapisu.

Możemy również przeprowadzić rzutowanie odwrotne, nazywane *rzutowaniem w dół* (ang. *downcasting*), ale powoduje to pewien problem, któremu przyjrzymy się w rozdziale następnym oraz w rozdziale „Informacje o typie".

Jeszcze o kompozycji i dziedziczeniu

W obiektowych językach programowania najbardziej prawdopodobnym sposobem tworzenia i użycia kodu, który zastosujesz, jest zwyczajne upakowanie razem danych i metod wewnątrz klasy oraz użycie obiektów tej klasy. Będziesz również wykorzystywał istniejące klasy, budując nowe poprzez kompozycję. Rzadziej przyjdzie Ci stosować dziedziczenie. Tak więc, chociaż na mechanizm dziedziczenia kładzie się spory nacisk podczas nauki programowania obiektowego, nie oznacza to, że trzeba stosować go wszędzie, gdzie tylko się da. Przeciwnie, dziedziczenie powinno się stosować oszczędnie — tylko wtedy, gdy jego użyteczność jest oczywista. Jednym z najlepszych sposobów, by ustalić, czy powinno się zastosować kompozycję czy dziedziczenie, jest pytanie, czy kiedykolwiek zajdzie potrzeba rzutowania w górę z nowej klasy do klasy bazowej. Jeżeli jest to konieczne, trzeba zastosować dziedziczenie, ale gdy rzutowanie w górę nie jest potrzebne, należy lepiej zastanowić się nad tym, czy rzeczywiście go potrzebujemy. Następny rozdział („Polimorfizm") będzie jednym z najbardziej przekonywujących powodów, by stosować rzutowanie w górę, ale jeśli zapamiętasz pytanie: „Czy potrzebuję rzutowania w górę?", będziesz dysponował dobrym narzędziem wspomagającym decyzję wyboru między kompozycją a dziedziczeniem.

Ćwiczenie 16. Napisz klasę płaza o nazwie `Amphibian` oraz dziedziczącą z niej klasę żaby — `Frog`. Zamieść odpowiednie metody w klasie bazowej. W funkcji `main()` stwórz obiekt klasy `Frog` i zrzutuj go na `Amphibian` — wykaż, że wszystkie metody nadal działają (2).

Ćwiczenie 17. Zmień ćwiczenie 16. tak, by przesłonić definicje metod klasy bazowej (dopisz nowe definicje metod o tej samej sygnaturze) w klasie `Frog`. Zaobserwuj, co się dzieje w `main()` (1).

Słowo kluczowe final

Słowo kluczowe `final` dostępne w Javie może mieć różne znaczenie w zależności od kontekstu, w jakim się go użyje, ale ogólnie oznacza: „To coś nie może być zmienione". Możemy bowiem chcieć uniemożliwić dokonywanie zmian z dwóch względów: projektowania lub wydajności. Ponieważ oba powody różnią się zasadniczo, słowo `final` może być stosowane niewłaściwie.

Przedstawię trzy sytuacje, w których można stosować `final`: dla zmiennych, metod i klas.

Zmienne finalne

W większości języków programowania istnieje sposób na to, by poinformować kompilator, że określona zmienna ma być „stała". Stałe są bowiem przydatne z dwóch powodów:

1. Mogą istnieć *stałe czasu kompilacji*, które nigdy nie będą zmieniane.
2. Mogą to być wartości inicjowane w czasie działania, których nie chcemy modyfikować.

W przypadku stałych czasu kompilacji kompilator ma możliwość umieszczenia wartości stałej wewnątrz obliczeń, w których jest ona użyta; oznacza to, że obliczenia mogą się wykonać w czasie kompilacji, eliminując pewne koszty podczas wykonania. W Javie stałe takiego rodzaju muszą być reprezentowane przez zmienne typu podstawowego, a wyraża się to właśnie poprzez użycie słowa kluczowego `final`. Wartość musi zostać przypisana w miejscu definicji stałej.

Pole, które jest zarówno statyczne, jak i finalne, zajmuje tylko jeden określony fragment pamięci i do tego nie może ulec zmianie.

Gdy stosujemy modyfikator `final` dla referencji do obiektów zamiast typów podstawowych, znaczenie modyfikatora może być nieco mylące. Dla typu elementarnego to *wartość* staje się stała, lecz dla referencji do obiektu to *referencja* zostaje stałą. Gdy zainicjujemy ją jako odwołanie do konkretnego obiektu, nigdy nie będzie można jej zmienić tak, aby wskazywała na jakiś inny obiekt. Jednak sam obiekt może być modyfikowany — w Javie nie istnieje sposób, aby uczynić jakiś obiekt niezmiennym (choć można napisać klasę tak, aby obiekty uzyskały efekt niezmienności). To samo ograniczenie dotyczy również tablic, które także są obiektami.

Oto przykład pokazujący pola finalne. Zauważ, że zgodnie z przyjętym stylem kodowania nazwy pól, które są zarówno finalne, jak i statyczne (to znaczy są stałymi czasu kompilacji), są pisane wielkimi literami, ze znakami podkreślenia pomiędzy członami składowymi:

```
//: reusing/FinalData.java
// Wpływ słowa "final" na składowe danych.
import java.util.*;
import static net.mindview.util.Print.*;

class Value {
  int i; // Dostęp pakietowy
  public Value(int i) { this.i = i; }
}

public class FinalData {
  private static Random rand = new Random(47);
  private String id;
  public FinalData(String id) { this.id = id; }
  // Mogą być stałymi czasu kompilacji:
  private final int valueOne = 9;
  private static final int VALUE_TWO = 99;
  // Typowa stała publiczna:
  public static final int VALUE_THREE = 39;
  // Nie mogą być stałymi czasu kompilacji:
  private final int i4 = rand.nextInt(20);
  static final int INT_5 = rand.nextInt(20);
  private Value v1 = new Value(11);
  private final Value v2 = new Value(22);
  private static final Value VAL_3 = new Value(33);
  // Tablice:
```

```
  private final int[] a = { 1, 2, 3, 4, 5, 6 };
  public String toString() {
    return id + ": " + "i4 = " + i4 + ", INT_5 = " + INT_5;
  }
  public static void main(String[] args) {
    FinalData fd1 = new FinalData("fd1");
    //! fd1.valueOne++; // Błąd: nie można zmieniać wartości
    fd1.v2.i++; // Obiekt nie jest stałą!
    fd1.v1 = new Value(9); // W porządku -- pole nie jest finalne
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // Obiekt nie jest stałą!
    //! fd1.v2 = new Value(0); // Błąd: nie da się
    //! fd1.VAL_3 = new Value(1); // Zmiana referencji
    //! fd1.a = new int[3];
    print(fd1);
    print("Tworzenie nowego obiektu FinalData");
    FinalData fd2 = new FinalData("fd2");
    print(fd1);
    print(fd2);
  }
} /* Output:
fd1: i4 = 15, INT_5 = 18
Tworzenie nowego obiektu FinalData
fd1: i4 = 15, INT_5 = 18
fd2: i4 = 13, INT_5 = 18
*///:~
```

Ponieważ `valueOne` oraz `VALUE_TWO`, będące typu podstawowego, są zadeklarowane jako finalne, mogą być użyte jako stałe już podczas kompilacji i tak właśnie jest. `VALUE_THREE` jest bardziej typowym sposobem definiowania takich stałych: publiczna — aby można ją stosować spoza pakietu, statyczna — aby zaznaczyć, że jest tylko jedna, i finalna — by określić, że to stała. Zauważ, że zmienne z modyfikatorami `final static` zainicjowane jako wartości stałe (oznacza to, że są stałymi czasu kompilacji) są z zasady nazywane dużymi literami, poprzez słowa oddzielone znakiem podkreślenia (dokładnie tak, jak stałe w C, gdzie konwencja ta powstała).

Z faktu, że coś jest finalne, nie wynika, że jego wartość jest znana podczas kompilacji. Widać to na przykładzie inicjalizacji zmiennych `i4` i `INT_5` poprzez inicjalizację w czasie wykonania z zastosowaniem generatora liczb losowych. Powyższy fragment kodu ukazuje również różnice między uczynieniem finalną zmiennej statycznej a egzemplarzowej. Różnica ujawnia się jedynie wtedy, gdy wartości są inicjowane w czasie działania, ponieważ wartości czasu kompilacji są traktowane przez kompilator tak samo (a najprawdopodobniej znikają w wyniku optymalizacji). Wartość zmiennej `i4` jest unikatowa dla obiektów `fd1` i `fd2`, natomiast wartość `INT_5` nie ulega zmianie po utworzeniu drugiego obiektu klasy `FinalData`. Dzieje się tak, gdyż jest to zmienna statyczna i jest inicjowana wyłącznie raz podczas ładowania, a nie za każdym razem, gdy tworzony jest obiekt.

Zmienne `v1` do `VAL_3` pokazują znaczenie referencji z modyfikatorem `final`. Jak widać w metodzie `main()`, z tego, że `v2` jest finalna, nie wynika, że nie można zmienić wartości obiektu. Nie można jednak przepiąć `v2` do nowego obiektu właśnie dlatego, że jest ona finalna. `v2` zawiera referencję, a zatem w tym przypadku słowo `final` oznacza, że nie można tej referencji przestawić. Jak widać, takie samo działanie występuje w przypadku tablic, które są po prostu innym rodzajem referencji (osobiście nie znam sposobu, by referencje

znajdujące się w tablicach uczynić same w sobie finalnymi). Stworzenie referencji finalnych zdaje się być zatem mniej przydatne niż stworzenie finalnych zmiennych typu podstawowego.

Ćwiczenie 18. Stwórz klasę zawierającą finalne pole statyczne oraz drugie pole finalne (niestatyczne) i pokaż różnicę pomiędzy nimi (2).

Puste zmienne finalne

Java pozwala na tworzenie *pustych zmiennych finalnych* będących polami zadeklarowanymi z modyfikatorem `final`, ale nie zainicjowanymi. W każdym razie zmienne takie *muszą* zostać zainicjowane przed użyciem i kompilator już tego dopilnuje. Puste zmienne finalne zapewniają znacznie większą elastyczność stosowania słowa `final` — na przykład finalne pola składowe mogą teraz być różne dla każdego z obiektów i w dodatku zachowują właściwość niezmienności. Oto przykład:

```
// : reusing/BlankFinal.java
// "Puste" pola finalne.

class Poppet {
  private int i;
  Poppet(int ii) { i = ii; }
}

public class BlankFinal {
  private final int i = 0; // Zainicjalizowane pole finalne
  private final int j; // Puste pole finalne
  private final Poppet p; // Pusta referencja finalna
  // Puste pola finalne MUSZĄ zostać zainicjalizowane w konstruktorze:
  public BlankFinal() {
    j = 1; // Inicjalizacja pustego pola finalnego
    p = new Poppet(1); // Inicjalizacja pustej referencji finalnej
  }
  public BlankFinal(int x) {
    j = x; // Inicjalizacja pustego pola finalnego
    p = new Poppet(x); // Inicjalizacja pustej referencji finalnej
  }
  public static void main(String[] args) {
    new BlankFinal();
    new BlankFinal(47);
  }
} ///:~
```

Kompilator wymusza przypisanie wartości zmiennym finalnym klasy bądź w miejscu definicji, bądź też w każdym z konstruktorów. Dzięki temu mamy gwarancję, że pola określone jako `final` są zawsze zainicjalizowane przed użyciem.

Ćwiczenie 19. Stwórz klasę z pustym finalnym odwołaniem do obiektu. Przeprowadź inicjalizację tego odwołania wewnątrz wszystkich konstruktorów. Wskaż sposób gwarantowania, że zmienne finalne muszą być zainicjowane przed użyciem oraz że nie mogą być zmieniane po zainicjowaniu (2).

Finalne argumenty

W Javie możliwe jest uczynienie finalnymi argumentów metody poprzez takie ich zadeklarowanie na liście argumentów. Oznacza to, że wewnątrz metody nie można zmienić wartości argumentów:

```
//: reusing/FinalArguments.java
// Słowo "final" przy argumentach metod.

class Gizmo {
  public void spin() {}
}

public class FinalArguments {
  void with(final Gizmo g) {
    //! g = new Gizmo(); // Niedozwolone -- g jest argumentem finalnym
  }
  void without(Gizmo g) {
    g = new Gizmo(); // W porządku -- g nie jest argumentem finalnym
    g.spin();
  }
  // void f(final int i) { i++; } // Nie można zmieniać
  // Finalny argument elementarny można tylko odczytywać:
  int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    FinalArguments bf = new FinalArguments();
    bf.without(null);
    bf.with(null);
  }
} ///:~
```

Metody `f()` i `g()` pokazują, co się dzieje, jeżeli argumenty są zmiennymi finalnymi typu podstawowego: można jedynie odczytać wartość, lecz nie można jej zmienić. Wykorzystuje się to przede wszystkim do przekazywania danych do nienazwanych klas wewnętrznych, o których powiemy sobie więcej w rozdziale „Klasy wewnętrzne".

Metody finalne

Są dwa powody, by zaznaczać metody jako finalne. Po pierwsze, jest to niejako założenie blokady na metodę, aby zabronić klasom pochodnym jakichkolwiek zmian — jest to podyktowane wymogami projektowymi, kiedy chcemy mieć pewność, że zachowanie metody pozostanie niezmienne i nie będzie mogło być przesłonięte.

Drugi powód stosowania metod finalnych jest związany z kwestią wydajności. Jeśli metoda zostanie finalna, pozwoli to kompilatorowi zamienić wywołania metody na *wywołania w miejscu* (ang. *inline*). Kiedy kompilator napotka wywołanie metody finalnej, może ominąć (dyskretnie) normalną procedurę włączenia kodu służącego do wywołania metody (położenie argumentów na stosie, skok do kodu metody i jego wykonanie, powrót, wyczyszczenie stosu z dołożonych argumentów oraz przekazanie wartości zwracanej) — zastępując takie wywołanie metody skopiowaniem jej rzeczywistego kodu. Dzięki temu eliminujemy narzut kosztów wywołania metody. Oczywiście, jeśli metoda jest duża, kod staje się rozbudowany i prawdopodobnie nie przyniesie to korzyści, gdyż jakakolwiek poprawa będzie bardzo mała w stosunku do czasu spędzonego w metodzie.

W nowszych wydaniach Javy maszyna wirtualna (chodzi zwłaszcza o technologię hotspot) potrafi wykryć takie sytuacje i w toku optymalizacji pozbyć się pośredniości, więc obecnie podpowiadanie kompilatorowi za pomocą słowa `final` jest niepotrzebne — a nawet programistów się do tego zniechęca. W Javie SE5 i SE6 należy zostawić kwestie wydajności kompilatorowi i maszynie wirtualnej, a metody oznaczać jako finalne tylko wtedy, jeśli ma to zapobiec ich przesłonięciu[1] [2].

final i private

Każda metoda prywatna klasy staje się pośrednio również finalna. Ponieważ nie ma dostępu do metod prywatnych, toteż nic można ich przesłonić (chociaż kompilator nie podaje komunikatu o błędzie, gdy się taką metodę nadpisuje, tak naprawdę nie jest to przesłonięcie, ale stworzenie nowej metody). Można dopisać modyfikator `final` obok `private`, ale niczego więcej się przez to nie uzyska.

Kwestia ta może wprowadzać w błąd z uwagi na to, że gdy próbujemy przesłonić metodę prywatną (która tym samym jest automatycznie finalna), wydaje się, że to działa (a kompilator nie zgłasza żadnych błędów):

```
// : reusing/FinalOverridingIllusion.java
// Pozornie można przesłonić metodę
// prywatną albo prywatną finalną.
import static net.mindview.util.Print.*;

class WithFinals {
  // Identyczna z metodą jedynie prywatną:
  private final void f() { print("WithFinals.f()"); }
  // Automatycznie również finalna:
  private void g() { print("WithFinals.g()"); }
}

class OverridingPrivate extends WithFinals {
  private final void f() {
    print("OverridingPrivate.f()");
  }
  private void g() {
    print("OverridingPrivate.g()");
  }
}

class OverridingPrivate2 extends OverridingPrivate {
  public final void f() {
    print("OverridingPrivate2.f()");
  }
  public void g() {
    print("OverridingPrivate2.g()");
  }
}
```

[1] Nie ulegaj chęci przedwczesnej optymalizacji programu. Jeśli system działa, lecz jest zbyt wolny, to możliwość usprawnienia jego pracy poprzez użycie słowa `final` jest wysoce wątpliwa. Informacje, które mogą pomóc w przyspieszaniu działania programów, można znaleźć w suplemencie publikowanym pod adresem *http://MindView.net/Books/BetterJava*.

[2] Innym sposobem, w jaki metody ze specyfikatorem `final` wpływają na wydajność, jest zysk z braku konieczności późnego wiązania —*przyp. red.*

```
public class FinalOverridingIllusion {
  public static void main(String[] args) {
    OverridingPrivate2 op2 = new OverridingPrivate2();
    op2.f();
    op2.g();
    // Można rzutować w górę:
    OverridingPrivate op = op2;
    // Ale nie można wywołać metod:
    //! op.f();
    //! op.g();
    // To samo tutaj:
    WithFinals wf = op2;
    //! wf.f();
    //! wf.g();
  }
} /* Output:
OverridingPrivate2.f()
OverridingPrivate2.g()
*///:~
```

„Przesłonięcie" może mieć miejsce jedynie wtedy, jeśli coś jest częścią interfejsu klasy bazowej. A więc musimy mieć możliwość rzutowania obiektu na klasę bazową i wywołania tej samej metody (wyjaśnię to dokładniej w kolejnym rozdziale). Jeżeli metoda jest prywatna, to nie zalicza się jej do interfejsu klasy bazowej. Jest to po prostu jakiś kod, który został ukryty wewnątrz klasy, i zdarzyło się, że posiada taką nazwę, ale jeśli w klasie pochodnej stworzymy metodę publiczną, chronioną lub dostępną w obrębie pakietu, to nie jest ona związana z tamtą metodą (choć otrzymała taką samą nazwę w klasie bazowej). A zatem w ten sposób nie przesłonimy metody, a jedynie stworzymy nową. Ponieważ metody prywatne są poza zasięgiem i są praktycznie niewidoczne, nie wpływa to na nic poza samą organizacją kodu klasy, w której została zdefiniowana.

Ćwiczenie 20. Pokaż, że zastosowanie adnotacji `@Override` pozwoliłoby uniknąć omawianego tu problemu (1).

Ćwiczenie 21. Napisz klasę zawierającą metodę finalną. Dopisz klasę pochodną tej klasy i spróbuj przesłonić metodę finalną (1).

Klasy finalne

Kiedy określimy całą klasę jako finalną (poprzez poprzedzenie jej definicji słowem kluczowym `final`), to zaznaczymy, że nie chcemy z niej dziedziczyć i również nie pozwalamy na to komukolwiek innemu. Innymi słowy, z jakichś powodów projekt klasy zakłada, że nigdy nie będzie potrzeby dokonywania żadnych zmian lub też ze względów bezpieczeństwa nie chcemy dalej dziedziczyć.

```
//: reusing/Jurassic.java
// Cała klasa jako klasa finalna.

class SmallBrain {}

final class Dinosaur {
  int i = 7;
  int j = 1;
```

```
  SmallBrain x = new SmallBrain();
  void f() {}
}

//! class Further extends Dinosaur {}
// Błąd: nie można dziedziczyć po finalnej klasie 'Dinosaur'

public class Jurassic {
  public static void main(String[] args) {
    Dinosaur n = new Dinosaur();
    n.f();
    n.i = 40;
    n.j++;
  }
} ///:~
```

Pola klasy mogą być finalne niezależnie od naszego wyboru. Ta sama zasada dotyczy stosowania modyfikatora `final` wobec składowych klasy, bez względu na to, czy klasa jest zdefiniowana jako finalna. Uczynienie klasy finalną uniemożliwia dalsze dziedziczenie — nic więcej. Jednak z uwagi na to, że nie można dziedziczyć, wszystkie metody klasy finalnej stają się również automatycznie finalne, nie ma bowiem możliwości ich przesłonięcia. Oczywiście można dopisać modyfikator `final` do metody w klasie finalnej, ale niczego przez to nie zyskamy.

Ćwiczenie 22. Napisz klasę finalną i spróbuj po niej dziedziczyć (1).

Ostrożnie z deklaracją final

Stosowanie metod finalnych podczas projektowania klasy wydaje się praktycznym rozwiązaniem. Można uważać, że wydajność jest bardzo istotna podczas używania klasy i że prawdopodobnie nikt nie będzie chciał przesłaniać naszych metod. Czasami jest to prawdą.

Przy takich założeniach lepiej być jednak ostrożnym. Przeważnie trudno jest przewidzieć, jak klasa może być wykorzystywana, szczególnie klasa ogólnego przeznaczenia. Jeśli określisz metodę jako finalną, to możesz zablokować możliwość powtórnego wykorzystania klasy poprzez dziedziczenie w kilku innych projektach z powodu braku wyobraźni, że może być w taki sposób wykorzystana.

Standardowa biblioteka Javy jest dobrym tego przykładem. Szczególnie klasa `Vector` z Java 1.0 i 1.1 była powszechnie stosowana i mogłaby być znacznie bardziej przydatna, gdyby pod pretekstem zwiększenia wydajności (które niemal na pewno było iluzoryczne) wszystkie jej metody nie były finalne. Łatwo sobie wyobrazić, że można chcieć dziedziczyć i przesłonić taką przydatną klasę, ale projektanci zdecydowali, że nie jest to właściwe rozwiązanie, co jest paradoksalne z dwóch powodów. Po pierwsze, klasa `Stack` dziedziczy po klasie `Vector`, co znaczy, *że jest* `Vectorem`, a to z logicznego punktu widzenia nie jest prawdą. Niemniej jednak to przypadek, kiedy sami projektanci języka postanowili dziedziczyć po klasie `Vector`: kiedy postanowili zaimplementować w ten sposób klasę `Stack`, przekonali się, że metody finalne są zbyt restrykcyjne.

Po drugie, wiele z najważniejszych metod klasy `Vector`, jak np. `addElement()` i `elementAt()`, jest synchronizowanych. Jak dowiesz się w rozdziale „Współbieżność", powoduje to znaczące zmniejszenie wydajności, co prawdopodobnie niweluje jakiekolwiek korzyści płynące z finalności. Potwierdza to pogląd, że programiści mało kiedy poprawnie odgadują pożądane miejsce optymalizacji. To bardzo niedobrze, że takie niezdarne projektowanie znajduje się w bibliotece standardowej i wszyscy musimy się z nim borykać (na szczęście współczesna biblioteka kontenerów zastąpiła klasę `Vector` znacznie bardziej cywilizowaną klasą `ArrayList`; niestety, do dziś programiści korzystają w swoich projektach z pierwotnej biblioteki kontenerów).

Warte odnotowania jest także to, że klasa `Hashtable` — inna ważna klasa z biblioteki standardowej — *nie* zawiera żadnych metod finalnych. Jak już wspominałem, jest oczywiste, że część klas była projektowana przez zupełnie kogo innego niż inne (przekonasz się, że nazwy metod `Hashtable` są znacznie krótsze w porównaniu z tymi z klasy `Vector`, co jest kolejnym tego dowodem). Sprawy te *nie* powinny być oczywiste dla użytkownika biblioteki klas. Jeśli istnieje niespójność, zmusza to użytkownika do wykonania większej pracy. Jest to jeszcze jeden czynnik wartościujący proces projektowania i wypuszczania kodu (biblioteka kontenerów z nowszych implementacji Javy zastąpiła klasę `Hashtable` przez `HashMap`).

Inicjalizacja i ładowanie klas

W językach bardziej tradycyjnych programy są ładowane w całości podczas startu. Później następuje inicjalizacja i wtedy program rozpoczyna działanie. Proces inicjalizacji w takich językach musi być starannie kontrolowany, aby kolejność inicjalizacji zmiennych statycznych nie sprawiała kłopotów. Przykładowo język C++ ma problemy, jeśli jedna zmienna statyczna dla swojej poprawnej konstrukcji wymaga pobrania wartości innej, zanim ta druga zostanie zainicjalizowana.

W Javie nie ma takiego problemu, gdyż mechanizm ładowania klas jest tu inny. To jedna z czynności upraszczanych podejściem polegającym na reprezentowaniu wszystkiego obiektami. Pamiętaj, że skompilowany kod każdej klasy jest przechowywany w oddzielnym pliku. Plik taki nie jest wczytywany, dopóki kod nie jest potrzebny. Można powiedzieć, że „kod klasy jest ładowany podczas pierwszego użycia". Zwykle jest to moment utworzenia pierwszego obiektu klasy, ale wczytanie kasy może też nastąpić wcześniej, w wyniku wywołania statycznej metody albo odwołania do statycznego pola klasy[3].

Czas pierwszego wykorzystania to również miejsce inicjalizacji zmiennych statycznych. Wszystkie obiekty statyczne oraz statyczne bloki kodu są podczas ładowania inicjalizowane w porządku tekstowym (to znaczy w kolejności, w jakiej zostały wpisane w definicji klasy). Oczywiście wszystko co statyczne jest inicjalizowane tylko raz.

[3] Konstruktor klasy również jest metodą statyczną, mimo braku stosownego modyfikatora w deklaracji. Ładowanie klasy następuje więc w momencie pierwszego odwołania do którejkolwiek statycznej składowej klasy.

Inicjalizacja w przypadku dziedziczenia

Przydatne będzie spojrzenie na proces inicjalizacji z włączeniem mechanizmu dziedziczenia, aby mieć pełny obraz tego, jak to działa. Zatem rozważmy następujący kod:

```
//: reusing/Beetle.java
// Proces inicjalizacji w pełnej krasie.
import static net.mindview.util.Print.*;

class Insect {
  private int i = 9;
  protected int j;
  Insect() {
    print("i = " + i + ", j = " + j);
    j = 39;
  }
  private static int x1 =
    printInit("static Insect.x1 zainicjalizowana");
  static int printInit(String s) {
    print(s);
    return 47;
  }
}

public class Beetle extends Insect {
  private int k = printInit("Beetle.k zainicjalizowana");
  public Beetle() {
    print("k = " + k);
    print("j = " + j);
  }
  private static int x2 =
    printInit("static Beetle.x2 zainicjalizowana");
  public static void main(String[] args) {
    print("Konstruktor klasy Beetle");
    Beetle b = new Beetle();
  }
} /* Output:
static Insect.x1 zainicjalizowana
static Beetle.x2 zainicjalizowana
Konstruktor klasy Beetle
i = 9, j = 0
Beetle.k zainicjalizowana
k = 47
j = 39
*///:~
```

Jeżeli uruchomimy interpreter Javy dla klasy `Beetle`, to najpierw odnajdzie metodę `Beetle.main()` (jest statyczna), dalej moduł ładujący będzie szukał skompilowanego kodu klasy `Beetle` (w pliku o nazwie *Beetle.class*). Podczas ładowania kodu moduł ładujący dostrzega, że ma ona klasę bazową (co określa słowo kluczowe `extends`), którą następnie wczytuje. Dzieje się to niezależnie od tego, czy tworzymy obiekt klasy bazowej czy nie (spróbuj wykomentować tworzenie obiektu, aby to sprawdzić).

Jeżeli klasa bazowa ma z kolei swoją klasę bazową, to ta druga też będzie ładowana itd. Następnie wykonywana jest inicjalizacja zmiennych statycznych głównej klasy bazowej (w tym przypadku klasy `Insect`) i dalej dzieje się to samo w klasie wywiedzionej itd.

Jest to istotne, gdyż inicjalizacja statyczna w klasach pochodnych może zależeć od właściwej inicjalizacji składowych klasy bazowej.

Teraz już wszystkie wymagane klasy są załadowane, a więc może nastąpić tworzenie obiektu. Najpierw wszystkie typy podstawowe w obiekcie są ustawiane na odpowiednie wartości domyślne, a referencje do obiektów na `null` — wszystko jednym płynnym ruchem, przez binarne wyzerowanie pamięci obiektu. Wtedy wywoływany jest konstruktor klasy bazowej. W naszym przypadku wywołanie jest automatyczne, ale można również określić takie wywołanie konstruktora klasy bazowej (jako pierwszą operację w konstruktorze klasy `Beetle()`), stosując słowo kluczowe `super`. Konstruktor klasy bazowej podlega temu samemu procesowi, w tym samym porządku, co konstruktor klasy wywiedzionej. Po zakończeniu działania konstruktora klasy bazowej inicjalizowane są zmienne egzemplarza, w kolejności wystąpienia w tekście. Na koniec wykonywana jest reszta ciała konstruktora.

Ćwiczenie 23. Udowodnij, że ładowanie klas ma miejsce tylko raz. Wykaż, że ładowanie może być spowodowane albo przez stworzenie pierwszego egzemplarza klasy, albo przez dostęp do składowej statycznej (2).

Ćwiczenie 24. W pliku *Beetle.java* dopisz klasę pochodną żuka z klasy `Beetle` o podobnej formie, jak istniejące tam klasy. Prześledź i zinterpretuj wynik działania (2).

Podsumowanie

Oba mechanizmy dziedziczenia i kompozycji pozwalają na stworzenie nowych typów danych z typów już istniejących. Kompozycję stosuje się w celu ponownego wykorzystania istniejących typów jako części wewnętrznej implementacji, a dziedziczenie, kiedy chcemy ponownie wykorzystać interfejs.

Przy dziedziczeniu klasa pochodna posiada też interfejs swojego przodka, może być więc *rzutowana w górę* do klasy bazowej, co ma — jak przekonasz się w kolejnym rozdziale — zasadnicze znaczenie dla polimorfizmu.

Mimo silnego nacisku na dziedziczenie w programowaniu obiektowym powinno się podczas pierwszego podejścia preferować kompozycję (ewentualnie delegację) i stosować dziedziczenie tylko wtedy, gdy okaże się naprawdę konieczne. Kompozycja bywa zazwyczaj bardziej elastyczna. Dodatkowo, stosując właściwości dziedziczenia, można zmienić właściwy typ obiektów składowych i tym sposobem zachowanie tych obiektów podczas pracy programu. Zatem można zmienić, w trakcie działania programu, zachowanie obiektu będącego kompozycją innych obiektów.

Podczas projektowania systemu naszym celem jest znalezienie lub stworzenie zbioru klas, w którym każda klasa ma określone zastosowanie i nie jest ani zbyt duża (obejmując tak wiele, że jest to nieporęczne do wykorzystania), ani też irytująco mała (nie można jej stosować samej lub bez dodatkowych funkcji). Jeśli projekt staje się zbyt zawiły, często pomaga wprowadzenie dodatkowych obiektów przez rozbicie istniejących na mniejsze.

Przygotowując się do projektowania systemu, należy mieć świadomość, że rozwijanie oprogramowania jest procesem przyrostowym, tak jak np. nauka. Całość bazuje na eksperymentach: możesz analizować założenia projektu tak długo jak zechcesz, ale mimo to, gdy przystąpisz do fazy właściwego tworzenia projektu, nie będziesz znał wszystkich odpowiedzi. Większe sukcesy i znacznie szybsze efekty osiągniesz, „hodując" projekt jak żywe i rozwijające się stworzenie, niż gdybyś chciał skonstruować go od razu w ostatecznej postaci, jak buduje się drapacz chmur. Dziedziczenie i kompozycja to dwa podstawowe narzędzia programowania obiektowego, których należy używać w owych eksperymentach.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 8.
Polimorfizm

„Zapytano mnie 'Panie Babbage, czy jeśli do maszyny włożysz złe liczby, dostaniesz prawidłowe odpowiedzi?'. Nie potrafię zrozumieć tego rodzaju pomylenia pojęć, które prowokuje do takich pytań".

— Charles Babbage (1791 – 1871).

Polimorfizm jest trzecim podstawowym składnikiem języka programowania zorientowanego obiektowo — zaraz po abstrakcji danych i dziedziczeniu.

Dostarcza on kolejną metodę separacji interfejsu od implementacji, oddzielania *co* od *jak*. Polepsza organizację i czytelność kodu oraz pozwala na tworzenie *rozszerzalnych* programów, które mogą być rozwijane nie tylko podczas powstawania, ale także później, gdy potrzebne są nowe możliwości.

Hermetyzacja umożliwia tworzenie nowych typów danych poprzez połączenie charakterystyki z określonym zestawem zachowań. Ukrywanie implementacji pozwala na oddzielenie tej ostatniej od interfejsu poprzez uczynienie szczegółów prywatnymi. Ten rodzaj mechanicznej organizacji ma sens dla kogoś z doświadczeniem w programowaniu proceduralnym, natomiast polimorfizm zajmuje się oddzielaniem w kategoriach *typów*. W poprzednim rozdziale zobaczyliśmy, w jaki sposób dziedziczenie pozwala traktować obiekt jako reprezentanta swego typu lub też typu bazowego. Możliwość taka ma ogromne znaczenie, ponieważ pozwala na traktowanie wielu typów (wywiedzionych ze wspólnego typu bazowego) jak jednego, dzięki czemu pojedynczy fragment kodu będzie działać tak samo na każdym z nich. Polimorficzne wywołanie metod umożliwia natomiast danemu typowi odróżnienie siebie od innych, podobnych, tak długo, jak długo są one wywiedzione ze wspólnego typu bazowego. Odróżnienie to polega na odmiennym działaniu metod, które mogą być wywoływane poprzez interfejs wspólnej klasy bazowej.

W tym rozdziale poznamy polimorfizm (zwany również *wiązaniem dynamicznym*, *późnym wiązaniem* lub *wiązaniem czasu wykonania*; ang. odpowiednio: *dynamic binding*, *late binding*, *run-time binding*), zaczynając od rzeczy najprostszych — przykładów, z których wycięto wszystko poza polimorficznym zachowaniem programu.

Rzutowanie w górę raz jeszcze

W poprzednim rozdziale obserwowaliśmy użycie obiektu jako reprezentanta swego typu lub jako obiektu typu bazowego. Traktowanie referencji do obiektu jako referencji do typu bazowego nazywane jest *rzutowaniem w górę* (ang. *upcasting*) z powodu sposobu, w jaki rysuje się diagramy przedstawiające drzewa dziedziczenia — z klasą bazową u góry.

Zaobserwowaliśmy również problem pokazany w poniższym programie dotyczącym instrumentów muzycznych.

Na początek powinniśmy zdefiniować wyliczenie `Note` reprezentujące nuty; jako że będzie wykorzystywane w większej liczbie przykładów, osadzimy je w osobnym pakiecie:

```
//: polymorphism/music/Note.java
// Nuty wygrywane na instrumentach muzycznych.
package polymorphism.music;

public enum Note {
    MIDDLE_C, C_SHARP, B_FLAT; // Itd.
} ///:~
```

Wyliczenia omawialiśmy w rozdziale „Inicjalizacja i sprzątanie".

W poniższym przykładzie obiekty `Wind` reprezentują pewien typ instrumentów, a zatem klasa `Wind` jest klasą pochodną klasy `Instrument`:

```
//: polymorphism/music/Instrument.java
package polymorphism.music;
import static net.mindview.util.Print.*;

class Instrument {
  public void play(Note n) {
    print("Instrument.play()");
  }
}
 ///:~

//: polymorphism/music/Wind.java
package polymorphism.music;

// Obiekty klasy Wind to obiekty typu Instrument,
// bo udostępniają identyczny interfejs:
public class Wind extends Instrument {
  // Ponowna definicja metody interfejsu:
  public void play(Note n) {
    System.out.println("Wind.play() " + n);
  }
} ///:~

//: polymorphism/music/Music.java
// Dziedziczenie i rzutowanie w górę.
package polymorphism.music;

public class Music {
```

```
  public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    tune(flute); // Rzutowanie w górę
  }
} /* Output:
Wind.play() MIDDLE_C
*///:~
```

Metoda `Music.tune()`akceptuje referencje typu `Instrument`, ale także wszystkich typów dziedziczących po `Instrument` — możemy to zaobserwować w metodzie `main()`, gdzie do `tune()` przekazywana jest referencja typu `Wind`, przy czym rzutowanie nie jest tutaj konieczne. Jest to dopuszczalne, ponieważ klasa `Wind` musi zawierać interfejs `Instrument`, ponieważ z niego dziedziczy. Rzutowanie w górę może „zawęzić" interfejs klasy, nie może jednak uczynić go mniejszym od pełnego interfejsu typu `Instrument`.

Zapominanie o typie obiektu

Program *Music.java* może się wydawać trochę dziwny. Czemu mielibyśmy celowo zapominać, jaki jest właściwy typ obiektu? To właśnie dzieje się przy rzutowaniu w górę, a wydaje się, że znacznie bardziej naturalnym byłoby, gdyby `tune()` pobierało po prostu jako swój argument referencję typu `Wind`. To prowadzi do kluczowego spostrzeżenia: jeśli tak zrobilibyśmy, potrzebowalibyśmy oddzielnej metody `tune()` dla każdego typu `Instrument` w naszym systemie. Przypuśćmy, że poszliśmy w tym kierunku i dodaliśmy instrumenty typów `Stringed` i `Brass`:

```
//: polymorphism/music/Music2.java
// Przeciążanie zamiast rzutowania w górę.
package polymorphism.music;
import static net.mindview.util.Print.*;

class Stringed extends Instrument {
  public void play(Note n) {
    print("Stringed.play() " + n);
  }
}

class Brass extends Instrument {
  public void play(Note n) {
    print("Brass.play() " + n);
  }
}

public class Music2 {
  public static void tune(Wind i) {
    i.play(Note.MIDDLE_C);
  }
  public static void tune(Stringed i) {
    i.play(Note.MIDDLE_C);
  }
  public static void tune(Brass i) {
    i.play(Note.MIDDLE_C);
```

```
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    Stringed violin = new Stringed();
    Brass frenchHorn = new Brass();
    tune(flute); // Bez rzutowania w górę
    tune(violin);
    tune(frenchHorn);
  }
} /* Output:
Wind.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
*///:~
```

Rozwiązanie takie działa, ma jednak poważną wadę: musimy napisać specjalną wersję metod dostosowaną do każdego nowego podtypu klasy `Instrument`, który dodamy. Oznacza to przede wszystkim więcej programowania na początku, ale również dużo pracy w przypadku, gdy będziemy chcieli dodać nową metodę w rodzaju `tune()` lub nowy `Instrument`. W połączeniu z faktem, że kompilator nie przekaże nam informacji o błędzie, gdy zapomnimy o przeciążeniu którejś z metod, sprawia to, że cały proces pracy z typami staje się niemożliwy do ogarnięcia.

Czy nie byłoby prostsze napisanie pojedynczej metody pobierającej jako argument obiekt klasy bazowej zamiast którejś z wyspecjalizowanych klas pochodnych? To znaczy: czy nie byłoby miło zapomnieć o istnieniu klas pochodnych i pisać kod komunikujący się jedynie z klasą bazową?

Dokładnie to umożliwia polimorfizm. Jednak wielu programistów (mających doświadczenie w programowaniu proceduralnym) ma pewne problemy ze zrozumieniem sposobu, w jaki on działa.

Ćwiczenie 1. Utwórz klasę `Cycle` z podklasami `Unicycle`, `Bicycle` i `Tricycle`. Pokaż, że egzemplarze typów pochodnych dają się rzutować w górę w metodzie `ride()` (2).

Mały trik

Trudność związaną z programem *Music.java* możemy zobaczyć w czasie jego uruchomienia. Na wyjściu pojawia się `Wind.play()`. Z pewnością jest to efekt pożądany, nie wiemy jednak, dlaczego tak się dzieje. Przyjrzyjmy się metodzie `tune()`:

```
public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
```

Otrzymuje ona referencję typu `Instrument`. Skąd zatem kompilator wie, że wskazuje ona na obiekt typu `Wind`, nie zaś `Stringed` albo `Brass`? Kompilator nie może tego wiedzieć. W celu lepszego zrozumienia tego tematu przydatne jest przyjrzenie się zagadnieniu *wiązania*.

Wiązanie wywołania metody

Połączenie wywołania metody z jej ciałem nazywamy *wiązaniem* (ang. *binding*). Gdy wiązanie dokonuje się przed wykonaniem programu (to znaczy jest wykonywane przez kompilator i linker, jeśli taki występuje), wtedy mówimy o *wczesnym wiązaniu* (ang. *early binding*). Mogłeś nie zetknąć się nigdy z tym terminem, ponieważ w językach proceduralnych on nie występował — na przykład w C stosowany jest tylko jeden tryb wywołania metody, a jest nim właśnie wiązanie wczesne.

Sprawiająca kłopoty część powyższego programu wymyka się wczesnemu wiązaniu, ponieważ kompilator nie może wiedzieć, jaką właściwie metodę powinien wywołać, mając jedynie referencję typu `Instrument`.

Rozwiązaniem jest tzw. *późne wiązanie* (ang. *late binding*). Nazwa ta oznacza, że wiązanie odbywa się w czasie wykonania programu i opiera się na właściwym typie obiektu. Proces ten jest także nazywany *wiązaniem dynamicznym* lub *wiązaniem czasu wykonania* (ang. *dynamic binding* i *run-time binding*). W języku implementującym późne wiązanie musi istnieć mechanizm określania typu obiektu w czasie wykonania programu w celu wywołania odpowiedniej metody. A zatem kompilator wciąż nie zna typu obiektu, jednak mechanizm wywołania metody sprawdza to i odwołuje się do właściwego jej ciała. W różnych językach odbywa się to w różny sposób, można sobie jednak wyobrazić, że pewna informacja o typie musi być wbudowana w obiekty.

Wszelkie wiązania w Javie są wiązaniami późnymi, chyba że metoda została zadeklarowana z użyciem modyfikatora `final` lub jest prywatna. Znaczy to, że normalnie nic musimy podejmować żadnych decyzji dotyczących używania późnego wiązania — występuje ono automatycznie.

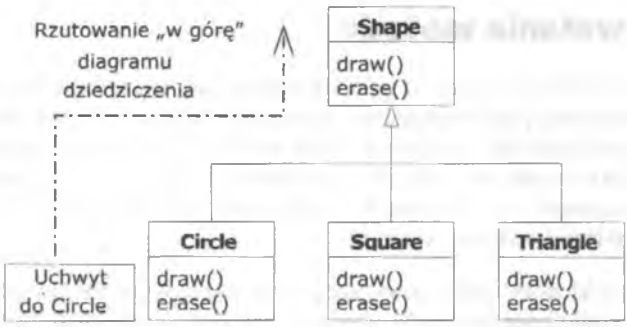
Dlaczego mielibyśmy deklarować metodę z użyciem słowa kluczowego `final`? Jak zauważyliśmy w poprzednim rozdziale, zapobiega to jej przesłonięciu przez kogoś innego. Być może jednak ważniejsze jest to, że właściwie wyłącza to późne wiązanie, a precyzyjniej, mówi kompilatorowi, że późne wiązanie nie jest konieczne. Pozwala mu to na wygenerowanie efektywniejszego kodu dla metod tak zadeklarowanych.

Uzyskiwanie poprawnego działania

Uświadomiwszy sobie, że wszelkie wywołania metod w Javie odbywają się polimorficznie z wykorzystaniem późnego wiązania, możemy zacząć pisać kod komunikujący się z klasą bazową, zakładając, iż wszelkie typy pochodne będą działać poprawnie z wykorzystaniem tego samego fragmentu kodu. Mówiąc innymi słowy: „Wysyłamy komunikat do obiektu i pozwalamy mu zdecydować, co naprawdę powinien zrobić".

Klasyczny w programowaniu zorientowanym obiektowo jest przykład „figur" — używa się go często, ponieważ jest łatwy w wizualizacji, choć może wprowadzić w błąd początkującego programistę, sugerując mu, że programowanie zorientowane obiektowo dotyczy jedynie grafiki, co nie jest oczywiście prawdą.

W przykładzie z figurami występują klasa bazowa o nazwie `Shape` („figura") i różne klasy pochodne — `Circle`, `Square`, `Triangle` itd., reprezentujące odpowiednio: okrąg, kwadrat oraz trójkąt. Przykład ten działa tak dobrze, ponieważ zdanie „okrąg jest rodzajem figury" jest zrozumiałe. Diagram dziedziczenia pokazuje te relacje:



Rzutowanie w górę może mieć miejsce w instrukcji tak prostej jak:

```
Shape s = new Circle();
```

Tworzony jest tutaj obiekt `Circle`, po czym jego referencja jest natychmiast przypisywana jako wartość typu `Shape`, co mogłoby wyglądać na błąd, jednak tak nie jest, ponieważ okrąg (`Circle`) *jest* figurą (`Shape`) poprzez dziedziczenie. Zatem kompilator przystaje na tę instrukcję i nie zgłasza komunikatu o błędzie.

Gdy wywołujemy jedną z metod klasy bazowej (które zostały przesłonięte w klasach pochodnych) poprzez:

```
s.draw();
```

możemy się spodziewać, że wywołana zostanie metoda `draw()` klasy `Shape` (ponieważ jest to w końcu referencja typu `Shape`), dlaczego więc kompilator miałby zrobić coś innego? A jednak wywoływana jest właściwa metoda `Circle.draw()` — dzieje się tak dzięki późnemu wiązaniu (polimorfizmowi).

Poniższy przykład pokazuje to w trochę inny sposób. Na początek utworzymy bibliotekę typów z rodziny `Shape`:

```
//: polymorphism/shape/Shape.java
package polymorphism.shape;

public class Shape {
  public void draw() {}
  public void erase() {}
} ///:~

//: polymorphism/shape/Circle.java
package polymorphism.shape;
import static net.mindview.util.Print.*;

public class Circle extends Shape {
  public void draw() { print("Circle.draw()"); }
  public void erase() { print("Circle.erase()"); }
} ///:~

//: polymorphism/shape/Square.java
package polymorphism.shape;
import static net.mindview.util.Print.*;
```

```
public class Square extends Shape {
  public void draw() { print("Square.draw()"); }
  public void erase() { print("Square.erase()"); }
} ///:~

//: polymorphism/shape/Triangle.java
package polymorphism.shape;
import static net.mindview.util.Print.*;

public class Triangle extends Shape {
  public void draw() { print("Triangle.draw()"); }
  public void erase() { print("Triangle.erase()"); }
} ///:~

//: polymorphism/shape/RandomShapeGenerator.java
// "Wytwórnia" tworząca losowe figury.
package polymorphism.shape;
import java.util.*;

public class RandomShapeGenerator {
  private Random rand = new Random(47);
  public Shape next() {
    switch(rand.nextInt(3)) {
      default:
      case 0: return new Circle();
      case 1: return new Square();
      case 2: return new Triangle();
    }
  }
} ///:~

//: polymorphism/Shapes.java
// Polimorfizm w Javie.
import polymorphism.shape.*;

public class Shapes {
  private static RandomShapeGenerator gen =
    new RandomShapeGenerator();
  public static void main(String[] args) {
    Shape[] s = new Shape[9];
    // Wypełnienie tablicy figurami:
    for(int i = 0; i < s.length; i++)
      s[i] = gen.next();
    // Polimorficzne wywołania metod:
    for(Shape shp : s)
      shp.draw();
  }
} /* Output:
Triangle.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Circle.draw()
*///:~
```

Klasa bazowa Shape ustanawia wspólny interfejs dla wszystkiego, co po niej dziedziczy — a zatem wszystkie figury mogą być rysowane i wymazywane. Wyprowadzone z niej klasy mogą przesłonić definicje tych metod, aby zapewnić unikatowe zachowanie dla każdego specyficznego typu figur.

Klasa `RandomShapeGenerator` jest swoistą „fabryką", która po każdym wywołaniu metody `next()` zwraca referencję losowo wybranej figury. Zauważmy, że rzutowanie w górę ma miejsce w każdej z instrukcji `return` pobierającej referencję typu: `Circle`, `Square` lub `Triangle` i zwracającej ją jako referencję typu zwracanego przez funkcję, a więc typu `Shape`. Zatem przy żadnym wywołaniu tej metody nie mamy szansy poznania rzeczywistego typu obiektu, ponieważ zawsze otrzymujemy referencję „gołego" typu `Shape`.

Metoda `main()` zawiera tablicę referencji typu `Shape`, wypełnioną poprzez wywołania `RandomShapeGenerator.next()`. W tym momencie wiemy, że mamy przedstawicieli klasy `Shape`, ale nic więcej. Kompilator też nie posiada innej wiedzy. Jednak jeśli przejdziemy przez tablicę i wywołamy `draw()` dla każdego elementu, wtedy w „magiczny" sposób ma miejsce właściwe, specyficzne dla typu zachowanie, które można zobaczyć w wynikach działania programu.

Oczywiście wyjście to może być inne przy każdym uruchomieniu programu, ponieważ figury wybierane są losowo. Celem losowego wybierania jest uwypuklenie faktu, że kompilator nie może posiadać żadnej specjalnej wiedzy umożliwiającej mu wykonanie odpowiednich wywołań w czasie kompilacji. Wszystkie wywołania `draw()` odbywają się poprzez dynamiczne wiązanie.

Ćwiczenie 2. Uzupełnij przykład z figurami adnotacjami `@Override` (1).

Ćwiczenie 3. Dodaj nową metodę wypisującą wiadomość do klasy bazowej w pliku *Shapes.java*, nie przesłaniaj jej jednak w klasie pochodnej. Wytłumacz, co się wtedy dzieje. Następnie przesłoń tę metodę w jednej z klas pochodnych (jednak nie we wszystkich) i zaobserwuj, co się dzieje. Na koniec przesłoń ją we wszystkich klasach pochodnych (1).

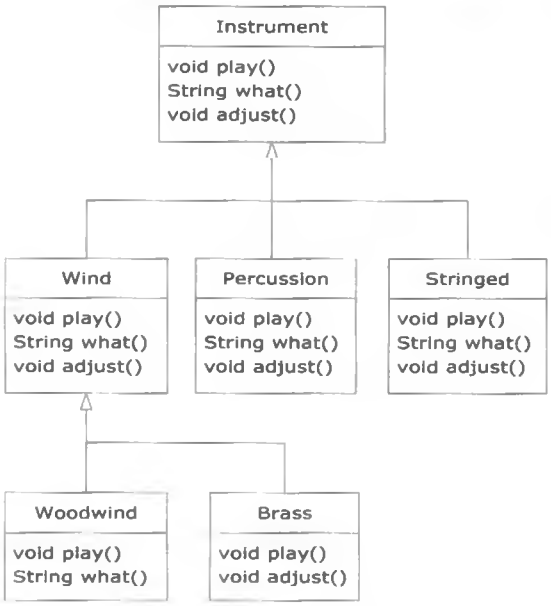
Ćwiczenie 4. Dodaj nową podklasę typu `Shape` do pliku *Shapes.java* i sprawdź w `main()`, że polimorfizm działa dla nowego typu tak samo, jak działał dla starych (2).

Ćwiczenie 5. Wróć do ćwiczenia 1. i uzupełnij klasę `Cycle` o metodę `wheels()`, która zwraca liczbę kół pojazdu. Zmień metodę `ride()` tak, aby wywoływała `wheels()`, i sprawdź działanie polimorfizmu w takim układzie (1).

Rozszerzalność

Powróćmy teraz do przykładu muzycznego. Dzięki polimorfizmowi możemy dodawać do systemu tyle typów, ile tylko chcemy, bez zmieniania metody `tune()`. W dobrze zaprojektowanym programie zorientowanym obiektowo większość (jeśli nie wszystkie) Twoich metod będzie powtarzać model metody `tune()` i komunikować się jedynie z interfejsem klasy bazowej. Program napisany w ten sposób jest *rozszerzalny*, ponieważ możliwe jest dodawanie nowych funkcji poprzez tworzenie nowych typów dziedziczących po wspólnej klasie bazowej. Metody manipulujące jedynie interfejsem klasy bazowej nie będą musiały być wcale zmieniane w celu zintegrowania z nowymi klasami.

Rozważmy, co stanie się z przykładem z instrumentami, jeśli dodamy nowe metody w klasie bazowej i pewną liczbę nowych klas. Oto stosowny diagram:



Wszystkie te nowe klasy działają poprawnie ze starą, niezmodyfikowaną metodą `tune()`. Nawet jeśli `tune()` znajdowałaby się w oddzielnym pliku, a do interfejsu klasy `Instrument` dodano by nowe metody, i tak `tune()` będzie działać poprawnie bez rekompilacji.

Oto implementacja powyższego diagramu:

```
// : polymorphism/music3/Music3.java
// Przykład programu rozszerzalnego.
package polymorphism.music3;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

class Instrument {
  void play(Note n) { print("Instrument.play() " + n); }
  String what() { return "Instrument"; }
  void adjust() { print("Strojenie obiektu Instrument"); }
}

class Wind extends Instrument {
  void play(Note n) { print("Wind.play() " + n); }
  String what() { return "Wind"; }
  void adjust() { print("Strojenie obiektu Wind"); }
}

class Percussion extends Instrument {
  void play(Note n) { print("Percussion.play() " + n); }
  String what() { return "Percussion"; }
  void adjust() { print("Strojenie obiektu Percussion"); }
}
```

```
class Stringed extends Instrument {
  void play(Note n) { print("Stringed.play() " + n); }
  String what() { return "Stringed"; }
  void adjust() { print("Strojenie obiektu Stringed"); }
}

class Brass extends Wind {
  void play(Note n) { print("Brass.play() " + n); }
  void adjust() { print("Strojenie obiektu Brass"); }
}

class Woodwind extends Wind {
  void play(Note n) { print("Woodwind.play() " + n); }
  String what() { return "Woodwind"; }
}

public class Music3 {
  // Metoda nie konkretyzuje typu, więc można z nią
  // stosować nowe typy dodawane do systemu:
  public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  public static void tuneAll(Instrument[] e) {
    for(Instrument i : e)
      tune(i);
  }
  public static void main(String[] args) {
    // Rzutowanie w górę w ramach wstawiania do tablicy:
    Instrument[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind()
    };
    tuneAll(orchestra);
  }
} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:~
```

Nowymi metodami są `what()` zwracająca obiekt typu `String` zawierający opis klasy oraz `adjust()` dostarczająca sposób dostrajania każdego z instrumentów.

W metodzie `main()` przy każdym umieszczeniu obiektu w tablicy `orchestra` dokonujemy automatycznie rzutowania tego typu w górę, do typu `Instrument`.

Widzimy, że metoda `tune()` ignoruje wszystkie zmiany kodu, które dokonały się wokoło niej, a mimo to działa prawidłowo. Właśnie to jest celem polimorfizmu. Zmiany kodu nie uszkadzają fragmentów programu, na które nie powinny mieć wpływu. Innymi słowy, polimorfizm jest jedną z najistotniejszych technik pozwalających programiście na „oddzielenie rzeczy, które się zmieniają, od rzeczy pozostających niezmiennymi".

Ćwiczenie 6. Zmień *Music3.java* tak, aby metoda `what()` stała się metodą `toString()` klasy `Object`. Spróbuj wypisać obiekty typu `Instrument`, wykorzystując `System.out.println()` (bez żadnego rzutowania) (1).

Ćwiczenie 7. Dodaj nowy `Instrument` do pliku *Music3.java* i sprawdź, czy polimorfizm działa z tym nowym typem (2).

Ćwiczenie 8. Zmodyfikuj *Music3.java* tak, aby obiekty `Instrument` były tworzone losowo, jak to się dzieje w *Shapes.java* (2).

Ćwiczenie 9. Stwórz hierarchię dziedziczenia dla `Gryzoni`: klasy `Mysz`, `Chomik` itd. W klasie bazowej umieść metody wspólne dla wszystkich `Gryzoni`, a następnie przesłoń je, realizując różnorodne zachowanie się klas pochodnych. Stwórz tablicę `Gryzoni` i wypełnij ją różnymi specyficznymi `Gryzoniami`, po czym wywołuj metody klasy bazowej, obserwując, co się dzieje (3).

Ćwiczenie 10. Stwórz klasę bazową z dwoma metodami. W pierwszej z metod wywołaj drugą z nich. Wywiedź poprzez dziedziczenie klasę pochodną ze stworzonej klasy bazowej i przesłoń drugą z metod. Stwórz obiekt klasy pochodnej, zrzutuj go w górę do klasy bazowej, a następnie wywołaj pierwszą z metod. Wytłumacz, co się dzieje (3).

Pułapka: „przesłanianie" metod prywatnych

Oto coś, co w nieświadomy sposób mógłbyś spróbować zrobić:

```
//: polymorphism/PrivateOverride.java
// Próba przesłonięcia metody prywatnej.
package polymorphism;
import static net.mindview.util.Print.*;

public class PrivateOverride {
  private void f() { print("prywatna metoda f()"); }
  public static void main(String[] args) {
    PrivateOverride po = new Derived();
    po.f();
  }
}

class Derived extends PrivateOverride {
  public void f() { print("publiczna metoda f()"); }
} /* Output:
prywatna metoda f()
*///:~
```

Można by oczekiwać, że program powinien wygenerować napis `publiczna metoda f()`, jednak metody prywatne automatycznie są także metodami finalnymi, przez co są niedostępne dla klas pochodnych. Oznacza to, że w powyższym przykładzie metoda `f()` klasy `Derived` jest całkowicie nową metodą — nie jest ona nawet przeciążona, gdyż metoda `f()` klasy bazowej nie jest dostępna w klasie `Derived`.

Jak zatem widać, metod prywatnych nie można przesłaniać. Należy się wystrzegać przesłaniania metod prywatnych, gdyż w takich przypadkach kompilator nie generuje żadnych błędów, a jednocześnie działanie programu nie jest zgodne z oczekiwaniami. W zasadzie w klasie pochodnej nie należy nadawać metodom nazw odpowiadających nazwom metod prywatnych klasy bazowej.

Pułapka: statyczne pola i metody

Po opanowaniu polimorfizmu zaczyna się popadać w skrajność, wszędzie widząc zachowania polimorficzne. Jednakże polimorficzne mogą być jedynie wywołania metod zwykłych, niestatycznych. Odwołanie do pola klasy będzie podlegać wiązaniu w czasie kompilacji, czego dowodzi następny przykład:

```
//: polymorphism/FieldAccess.java
// Bezpośrednie odwołania do pól są wiązane w czasie kompilacji.

class Super {
  public int field = 0;
  public int getField() { return field; }
}

class Sub extends Super {
  public int field = 1;
  public int getField() { return field; }
  public int getSuperField() { return super.field; }
}

public class FieldAccess {
  public static void main(String[] args) {
    Super sup = new Sub(); // Rzutowanie w górę
    System.out.println("sup.field = " + sup.field +
      ", sup.getField() = " + sup.getField());
    Sub sub = new Sub();
    System.out.println("sub.field = " +
      sub.field + ", sub.getField() = " +
      sub.getField() +
      ", sub.getSuperField() = " +
      sub.getSuperField());
  }
} /* Output:
sup.field = 0, sup.getField() = 1
sub.field = 1, sub.getField() = 1, sub.getSuperField() = 0
*///:~
```

Kiedy obiekt klasy Sub jest rzutowany w górę na postać referencji Super, wszelkie odwołania do pól są rozstrzygane przez kompilator, a więc nie przejawiają polimorfizmu. W tym przykładzie dla składowych `Super.field` i `Sub.field` przydzielone zostały odrębne obszary pamięci. Dlatego obiekt Sub zawiera w zasadzie dwa pola o nazwie `field`: własne i pole z klasy bazowej Super. Jednak przy odwołaniu do pola `field` w klasie Sub domyślnym polem jest nie pole klasy Super, tylko właśnie pole klasy Sub. Aby odwołać się do pola z klasy bazowej, należy skorzystać z odwołania `super.field`.

Choć wydaje się to kłopotliwe i niespójne, w praktyce problem w zasadzie nie istnieje. Przede wszystkim wszelkie pola są zazwyczaj oznaczane jako prywatne, więc bezpośrednie odwołania do nich i tak są niemożliwe — pola są udostępniane jako efekty uboczne wywołań metod. Do tego mało kto nadaje polom w klasach bazowych i pochodnych identyczne nazwy, bo jest to bardzo mylące.

Metody statyczne nie przejawiają polimorfizmu:

```
//: polymorphism/StaticPolymorphism.java
// Metody statyczne nie przejawiają polimorfizmu.

class StaticSuper {
  public static String staticGet() {
    return "Bazowa wersja staticGet()";
  }
  public String dynamicGet() {
    return "Bazowa wersja dynamicGet()";
  }
}

class StaticSub extends StaticSuper {
  public static String staticGet() {
    return "Pochodna wersja staticGet()";
  }
  public String dynamicGet() {
    return "Pochodna wersja dynamicGet()";
  }
}

public class StaticPolymorphism {
  public static void main(String[] args) {
    StaticSuper sup = new StaticSub(); // Rzutowanie w górę
    System.out.println(sup.staticGet());
    System.out.println(sup.dynamicGet());
  }
} /* Output:
Bazowa wersja staticGet()
Pochodna wersja dynamicGet()
*///:~
```

Metody statyczne są skojarzone z klasami, a nie z pojedynczymi obiektami.

Konstruktory a polimorfizm

Konstruktory jak zwykle różnią się od innych rodzajów metod także w odniesieniu do polimorfizmu. Mimo że konstruktory nie są polimorficzne (w rzeczywistości są to metody statyczne, choć deklaracja `static` jest w ich przypadku niejawna), istotne jest zrozumienie sposobu, w jaki współdziałają z tym mechanizmem w złożonych hierarchiach klas. Pomoże to uniknąć wielu nieprzyjemnych niespodzianek.

Kolejność wywołań konstruktorów

Kolejność wywoływania konstruktorów została częściowo omówiona w rozdziale „Inicjalizacja i sprzątanie" i ponownie w rozdziale „Wielokrotne wykorzystanie klas", jednak było to jeszcze przed wprowadzeniem pojęcia dziedziczenia.

Konstruktor klasy bazowej jest wywoływany zawsze w konstruktorze klasy pochodnej, co powoduje łańcuchowe przejście przez hierarchię dziedziczenia, dzięki czemu wywoływany jest konstruktor każdej klasy bazowej. Jest to rozwiązanie rozsądne, ponieważ zadanie konstruktora jest specjalne: dopilnowanie, aby obiekt został stworzony poprawnie. Klasy pochodne mają dostęp jedynie do własnych składników, przeważnie nie mają zaś go do składników klasy bazowej (będących zwykle prywatnymi). Jedynie konstruktor klasy bazowej posiada wiedzę niezbędną do zainicjalizowania tych składników. Podstawowe znaczenie ma zatem to, aby wszystkie konstruktory zostały wywołane, w przeciwnym razie cały obiekt nie zostałby poprawnie skonstruowany. Z tego właśnie powodu kompilator wymusza wywołanie konstruktora dla każdej części klasy pochodnej. Jeżeli nie wywołamy jawnie konstruktora klasy bazowej (w ciele konstruktora klasy pochodnej), wtedy po cichu zostanie wywołany jej konstruktor domyślny. Jeżeli klasa bazowa nie ma konstruktora domyślnego, kompilator zaprotestuje (jeżeli klasa nie posiada żadnych konstruktorów, konstruktor domyślny jest tworzony automatycznie przez kompilator).

Przyjrzyjmy się przykładowi pokazującemu wpływ kompozycji, dziedziczenia i polimorfizmu na porządek konstrukcji obiektów:

```
//: polymorphism/Sandwich.java
// Kolejność wywołań destruktorów.
package polymorphism;
import static net.mindview.util.Print.*;

class Meal {
  Meal() { print("Meal()"); }
}

class Bread {
  Bread() { print("Bread()"); }
}

class Cheese {
  Cheese() { print("Cheese()"); }
}

class Lettuce {
  Lettuce() { print("Lettuce()"); }
}

class Lunch extends Meal {
  Lunch() { print("Lunch()"); }
}

class PortableLunch extends Lunch {
  PortableLunch() { print("PortableLunch()");}
}

public class Sandwich extends PortableLunch {
  private Bread b = new Bread();
  private Cheese c = new Cheese();
  private Lettuce l = new Lettuce();
  public Sandwich() { print("Sandwich()"); }
  public static void main(String[] args) {
    new Sandwich();
  }
```

```
} /* Output:
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
*///:~
```

W tym przykładzie tworzymy klasę złożoną z innych klas, a każda z nich przedstawia się w konstruktorze. Ważną klasą jest `Sandwich`, odzwierciedla bowiem trzy poziomy dziedziczenia (lub cztery, jeśli weźmiemy pod uwagę niejawne dziedziczenia po klasie `Object`) oraz zawiera trzy obiekty składowe. Wyniki programu przedstawiają rezultaty utworzenia obiektu `Sandwich` w metodzie `main()`. A zatem porządek konstrukcji złożonego obiektu jest następujący:

1. Wywoływany jest konstruktor klasy bazowej. Krok ten jest powtarzany rekursywnie tak, że najpierw konstruowany jest korzeń hierarchii, potem pierwsza klasa pochodna itd., aż osiągnięta zostanie najniższa w hierarchii klasa pochodna.
2. Następuje inicjalizacja składowych w kolejności, w jakiej zostały zadeklarowane.
3. Wykonywane jest ciało konstruktora klasy pochodnej.

Kolejność wywołań konstruktorów jest istotna. Dziedzicząc, wiemy wszystko o klasie bazowej oraz mamy dostęp do wszystkich jej składowych publicznych i chronionych. Znaczy to, że musimy mieć możliwość założenia, że wszystkie składowe klas bazowych mają już poprawne wartości, gdy odwołujemy się do nich z klasy pochodnej. W przypadku wykonywania normalnej metody zakłada się, że konstrukcja obiektu miała miejsce wcześniej, a zatem wszystkie składowe zostały już stworzone. Jedynym sposobem zagwarantowania tego jest wywoływanie najpierw konstruktora klasy bazowej. Potem, będąc w konstruktorze klasy pochodnej, wiemy, że składowe klasy bazowej, do których mamy dostęp, zostały już zainicjalizowane poprawnie. Posiadanie wewnątrz konstruktora „wiedzy, że składowe zostały poprawnie zainicjalizowane" jest także powodem, dla którego — gdzie tylko się da — powinno się inicjalizować obiekty składowe (a więc obiekty umieszczone poprzez kompozycję) w miejscu ich definicji w klasie (np. `b`, `c` i `l` w powyższym przykładzie). Stosując się do tych zaleceń, możemy mieć pewność, że wszystkie składowe klasy bazowej *oraz* obiekty składowe zostały zainicjalizowane. Niestety, jak zobaczymy za chwilę, nie jest to możliwe w każdym przypadku.

Ćwiczenie 11. Dodaj do pliku *Sandwich.java* klasę o nazwie `Pickle` (1).

Dziedziczenie a sprzątanie

Tworząc nową klasę z użyciem kompozycji i dziedziczenia, nie martwimy się nigdy o sprzątanie jej składowych — zazwyczaj zadanie to można pozostawić odśmiecaczowi. Jeśli jednak musimy wykonać je samodzielnie, należy zachować ostrożność i w każdej z nowych klas stworzyć metody `dispose()` (ja akurat wybrałem tę nazwę, choć może ona być całkowicie dowolna). Wykorzystując dziedziczenie, musimy jednak przesłonić w klasie pochodnej metodę `dispose()`, jeżeli tylko mamy do wykonania jakieś specjalne sprzątanie, które powinno zostać przeprowadzone jako część odśmiecania. Gdy przesłaniamy

dispose() w klasie pochodnej, musimy pamiętać o wywołaniu wersji tej metody z klasy bazowej, ponieważ w przeciwnym razie sprzątanie klasy bazowej nie odbędzie się. Dowodzi tego poniższy przykład:

```
//: polymorphism/Frog.java
// Dziedziczenie a sprzątanie.
package polymorphism;
import static net.mindview.util.Print.*;

class Characteristic {
  private String s;
  Characteristic(String s) {
    this.s = s;
    print("Tworzenie cechy (Characteristic) " + s);
  }
  protected void dispose() {
    print("Usuwanie cechy (Characteristic) " + s);
  }
}

class Description {
  private String s;
  Description(String s) {
    this.s = s;
    print("Tworzenie opisu (Description) " + s);
  }
  protected void dispose() {
    print("Usuwanie opisu (Description) " + s);
  }
}

class LivingCreature {
  private Characteristic p =
    new Characteristic("żyje");
  private Description t =
    new Description("Stworzenie żyjące (LivingCreature)");
  LivingCreature() {
    print("LivingCreature()");
  }
  protected void dispose() {
    print("Usuwanie LivingCreature ");
    t.dispose();
    p.dispose();
  }
}

class Animal extends LivingCreature {
  private Characteristic p =
    new Characteristic("ma serce");
  private Description t =
    new Description("Zwierzę (Animal), nie roślina");
  Animal() { print("Animal()"); }
  protected void dispose() {
    print("Usuwanie Animal");
    t.dispose();
    p.dispose();
    super.dispose();
  }
}
```

```
class Amphibian extends Animal {
  private Characteristic p =
    new Characteristic("może żyć w wodzie");
  private Description t =
    new Description("Woda i lądy (Amphibian)");
  Amphibian() {
    print("Amphibian()");
  }
  protected void dispose() {
    print("Usuwanie Amphibian");
    t.dispose();
    p.dispose();
    super.dispose();
  }
}

public class Frog extends Amphibian {
  private Characteristic p = new Characteristic("rechocze");
  private Description t = new Description("Owadożerne");
  public Frog() { print("Frog()"); }
  protected void dispose() {
    print("Usuwanie Frog");
    t.dispose();
    p.dispose();
    super.dispose();
  }
  public static void main(String[] args) {
    Frog frog = new Frog();
    print("Pa!");
    frog.dispose();
  }
} /* Output:
Tworzenie cechy (Characteristic) żyje
Tworzenie opisu (Description) Stworzenie żyjące (LivingCreature)
LivingCreature()
Tworzenie cechy (Characteristic) ma serce
Tworzenie opisu (Description) Zwierzę (Animal), nie roślina
Animal()
Tworzenie cechy (Characteristic) może żyć w wodzie
Tworzenie opisu (Description) Woda i lądy (Amphibian)
Amphibian()
Tworzenie cechy (Characteristic) rechocze
Tworzenie opisu (Description) Owadożerne
Frog()
Pa!
Usuwanie Frog
Usuwanie opisu (Description) Owadożerne
Usuwanie cechy (Characteristic) rechocze
Usuwanie Amphibian
Usuwanie opisu (Description) Woda i lądy (Amphibian)
Usuwanie cechy (Characteristic) może żyć w wodzie
Usuwanie Animal
Usuwanie opisu (Description) Zwierzę (Animal), nie roślina
Usuwanie cechy (Characteristic) ma serce
Usuwanie LivingCreature
Usuwanie opisu (Description) Stworzenie żyjące (LivingCreature)
Usuwanie cechy (Characteristic) żyje
*///:~
```

Każda klasa hierarchii zawiera składowe: obiekt klas `Characteristic` oraz `Description`, które także należy odpowiednio usunąć. Sprzątnie tych obiektów powinno odbywać się w kolejności odwrotnej od porządku ich tworzenia, co stanowi zabezpieczenie przed sytuacją, w której jeden z podobiektów jest zależy od drugiego. W odniesieniu do pól oznacza to, że powinny one być usuwane w odwrotnej kolejności od ich deklarowania (ponieważ inicjalizacja pól odbywa się w kolejności, w jakiej zostały one zadeklarowane). W przypadku klas bazowych należy najpierw posprzątać klasy pochodne, a dopiero potem samą klasę bazową (zgodnie z rozwiązaniem przyjętym w destruktorach stosowanych w C++). Wynika to z faktu, że podczas sprzątania klas pochodnych mogą być wywoływane metody klasy bazowej, wymagające istnienia pewnych komponentów klasy bazowej; a zatem komponentów tych nie można usuwać zbyt wcześnie. Analizując wyniki wykonania programu, można się przekonać, że wszystkie części obiektu `Frog` są usuwane w odwrotnej kolejności od ich tworzenia.

Przykład ten pokazuje, że choć nie zawsze trzeba po sobie „sprzątać", jeśli już to robimy, należy uważać.

Ćwiczenie 12. Zmodyfikuj ćwiczenie 9. tak, aby demonstrowało kolejność inicjalizacji klas bazowych i pochodnych. Następnie dodaj obiekty składowe zarówno do klasy bazowej, jak i pochodnej, po czym sprawdź, w jakiej kolejności następuje ich inicjalizacja w trakcie konstrukcji (3).

Zauważ też, że obiekt klasy `Frog` posiada swoje obiekty składowe „na własność". Tworzy je, a ponieważ wie, jak długo są potrzebne (tak długo, jak długo istnieje obiekt `Frog`), wie więc, kiedy wywołać na ich rzecz ich metody `dispose()`. Gdyby jednak jeden z takich obiektów składowych był wspólny dla kilku obiektów `Frog`, problem znacznie by się skomplikował, bo nie można byłoby założyć prostego wywołania `dispose()`. W takim przypadku konieczne mogłoby się okazać zliczanie referencji w celu ustalenia liczby obiektów odwołujących się do obiektu wspólnego. Wyglądałoby to tak:

```
//: polymorphism/ReferenceCounting.java
// Sprzątanie wspólnych obiektów składowych
import static net.mindview.util.Print.*;

class Shared {
  private int refcount = 0;
  private static long counter = 0;
  private final long id = counter++;
  public Shared() {
    print("Tworzenie " + this);
  }
  public void addRef() { refcount++; }
  protected void dispose() {
    if(--refcount == 0)
      print("Usuwanie " + this);
  }
  public String toString() { return "Shared " + id; }
}

class Composing {
  private Shared shared;
  private static long counter = 0;
  private final long id = counter++;
  public Composing(Shared shared) {
```

```
    print("Tworzenie " + this);
    this.shared = shared;
    this.shared.addRef();
  }
  protected void dispose() {
    print("Usuwanie " + this);
    shared.dispose();
  }
  public String toString() { return "Composing " + id; }
}

public class ReferenceCounting {
  public static void main(String[] args) {
    Shared shared = new Shared();
    Composing[] composing = { new Composing(shared),
      new Composing(shared), new Composing(shared),
      new Composing(shared), new Composing(shared) };
    for(Composing c : composing)
      c.dispose();
  }
} /* Output:
Tworzenie Shared 0
Tworzenie Composing 0
Tworzenie Composing 1
Tworzenie Composing 2
Tworzenie Composing 3
Tworzenie Composing 4
Usuwanie Composing 0
Usuwanie Composing 1
Usuwanie Composing 2
Usuwanie Composing 3
Usuwanie Composing 4
Usuwanie Shared 0
*///:~
```

Statyczna składowa typu podstawowego `long` o nazwie `counter` rejestruje liczbę utworzonych egzemplarzy klasy `Shared`, a także stanowi wartość identyfikatora `id`. Typ licznika obiektów to `long` zamiast `int` — ma to zapobiec przepełnieniu licznika (to przejaw promowania dobrych praktyk programistycznych; prawdopodobieństwo „zawinięcia" wartości licznika obiektów w przykładach z tej książki jest zerowe). Składowa `id` to składowa finalna, ponieważ nie spodziewamy się zmian jej wartości w czasie życia obiektu.

Przy kojarzeniu obiektu wspólnego z własną klasą należy pamiętać o wywołaniu jego metody `addRef()`, natomiast o przeprowadzeniu sprzątania zdecyduje metoda `dispose()` monitorująca licznik referencji. Technika ta wymaga pewnej konsekwencji i pilności, ale przy użytkowaniu obiektów wspólnych wymagających operacji porządkowych alternatyw w zasadzie nie ma.

Ćwiczenie 13. Dodaj do pliku *ReferenceCounting.java* metodę `finalize()` weryfikującą warunek zakończenia (zobacz rozdział „Inicjalizacja i sprzątanie") (3).

Ćwiczenie 14. Zmień ćwiczenie 12. tak, aby jeden z obiektów składowych był obiektem wspólnym ze zliczaniem odwołań; wykaż, że zliczanie działa poprawnie (4).

Zachowanie metod polimorficznych wewnątrz konstruktorów

Hierarchia wywołań konstruktorów powoduje powstanie ciekawego dylematu. Co się dzieje, jeżeli jesteśmy wewnątrz konstruktora i wywołamy dynamicznie wiązaną metodę konstruowanego obiektu?

W obrębie zwykłej metody wiadomo, co się stanie — dynamiczne wiązanie zostanie poprawnie zinterpretowane dopiero w czasie wykonania, ponieważ pisząc metodę, nie możemy wiedzieć, czy zostanie ona wywołana na rzecz klasy, w której ją piszemy, czy też dla jakiejś klasy pochodnej.

Jest jednak nieco inaczej. Gdy wywołujemy metodę dynamicznie wiązaną w konstruktorze, wtedy używana jest przesłonięta wersja tej metody. *Rezultat* może być jednak dość niespodziewany, bo przesłonięta metoda zostanie wywołana jeszcze przed zakończeniem konstrukcji obiektu. Może to stanowić przyczynę wielu trudnych do wychwycenia błędów.

Z koncepcyjnego punktu widzenia zadaniem konstruktora jest zapoczątkowanie istnienia obiektu (co nie jest zwykłym działaniem). Wewnątrz konstruktorów obiekty mogą być jedynie częściowo stworzone — wiemy jedynie, że zainicjalizowane zostały obiekty klas bazowych. Jeśli konstruktor jest tylko elementem konstrukcji obiektu klasy wyprowadzonej z klasy tego konstruktora, to części pochodne są w momencie wywołania konstruktora klasy bieżącej jeszcze niezainicjalizowane. Dynamiczne wiązanie metody sięga jednak w dół hierarchii dziedziczenia — wywoływana jest metoda klasy pochodnej. Robiąc coś takiego w konstruktorze, wywołujemy metodę mogącą manipulować składowymi jeszcze nie zainicjalizowanymi — to prosta recepta na katastrofę.

Możemy zaobserwować ten problem w poniższym przykładzie:

```
//: polymorphism/PolyConstructors.java
// Konstruktory a polimorfizm
// - nieoczekiwane zachowanie.
import static net.mindview.util.Print.*;

class Glyph {
  void draw() { print("Glyph.draw()"); }
  Glyph() {
    print("Glyph() przed draw()");
    draw();
    print("Glyph() po draw()");
  }
}

class RoundGlyph extends Glyph {
  private int radius = 1;
  RoundGlyph(int r) {
    radius = r;
    print("RoundGlyph.RoundGlyph(), radius = " + radius);
  }
  void draw() {
    print("RoundGlyph.draw(), radius = " + radius);
  }
}
```

```
public class PolyConstructors {
  public static void main(String[] args) {
    new RoundGlyph(5);
  }
} /* Output:
Glyph() przed draw()
RoundGlyph.draw(), radius = 0
Glyph() po draw()
RoundGlyph.RoundGlyph(), radius = 5
*///:~
```

Metoda `Glyph.draw()` jest przeznaczona do przesłaniania, co następuje w klasie `RoundGlyph`. Jednakże metodę tę wywołuje konstruktor klasy `Glyph`, a wywołanie to ostatecznie powoduje wywołanie `RoundGlyph.draw()`, co wydaje się być zgodne z zamierzeniami. Jednakże analizując wyniki wygenerowane przez program, można zauważyć, że w chwili gdy konstruktor klasy `Glyph` wywołuje metodę `draw()`, składowa `radius` nie ma nawet swej domyślnej wartości początkowej 1. Jest równa 0. W rezultacie narysowana zostanie kropka albo nic, a my będziemy się temu przypatrywać, próbując odgadnąć, dlaczego program nie działa.

Kluczem do rozwiązania tej zagadki jest to, że porządek inicjalizacji opisany wcześniej nie jest kompletny. W rzeczywistości proces inicjalizacji przebiega następująco:

1. Obiektom przydzielana jest pamięć. Zanim stanie się cokolwiek innego, jest ona dodatkowo wypełniana zerami.
2. Wywoływane są konstruktory klas bazowych, jak opisano poprzednio. W powyższym przykładzie to właśnie w tym punkcie wywoływana jest przesłonięta wersja metody `draw()` (tak właśnie, dzieje się to przed wywołaniem konstruktora klasy `RoundGlyph`), która odkrywa, że wartością `radius` jest 0, co jest spowodowane przez krok 1.
3. Inicjalizatory obiektów składowych są wywoływane w kolejności deklaracji tychże składowych.
4. Wykonywane jest ciało konstruktora klasy pochodnej.

Proces taki ma pewną zaletę polegającą na tym, że wszystko jest w najgorszym przypadku inicjalizowane na zero (lub na to, co oznacza 0 w przypadku konkretnego typu danych). Obejmuje to w szczególności referencje do obiektów osadzone w klasie poprzez kompozycję, które są ustawiane na `null`. Jeżeli zatem zapomnimy o inicjalizacji referencji, otrzymamy wyjątek czasu wykonania. Cała reszta staje się zerami, które są zwykle wartościami rzucającymi się w oczy przy obserwowaniu wyjścia programu.

Z drugiej strony, rezultat działania tego programu powinien być dla nas raczej przerażający. Zrobiliśmy coś całkowicie logicznego, a mimo to zachowanie programu jest w tajemniczy sposób niepoprawne, bez żadnych skarg ze strony kompilatora (C++ zachowuje się w takiej sytuacji znacznie rozsądniej). Błędy tego typu mogą pozostać w ukryciu bardzo długo.

W rezultacie dobrą wskazówką w przypadku konstruktorów jest: „Zrób tak mało, jak to tylko możliwe, aby ustawić obiekt w poprawnym stanie, a jeśli się da, unikaj wywoływania metod". Jedynymi metodami, które można bezpiecznie wywoływać z konstruktorów,

są te zadeklarowane z użyciem `final` (odnosi się to również do metod prywatnych, ponieważ stają się one ostateczne automatycznie). Nie mogą być one przesłaniane, a przez to nie mogą zrobić opisanej powyżej przykrej niespodzianki. Ta wytyczna nie zawsze da się utrzymać, ale warto o niej pamiętać.

Ćwiczenie 15. Dodaj do pliku *PolyContructors.java* klasę `RectangularGlyph` i zademonstruj problem opisany w tym punkcie (2).

Kowariancja typów zwracanych

Java SE5 uzupełnia język o *kowariantne typy zwracane*, co oznacza, że przesłonięta metoda klasy pochodnej może zwracać typ pochodny względem typu zwracanego przez metodę bazową:

```
//: polymorphism/CovariantReturn.java

class Grain {
  public String toString() { return "Grain"; }
}

class Wheat extends Grain {
  public String toString() { return "Wheat"; }
}

class Mill {
  Grain process() { return new Grain(); }
}

class WheatMill extends Mill {
  Wheat process() { return new Wheat(); }
}

public class CovariantReturn {
  public static void main(String[] args) {
    Mill m = new Mill();
    Grain g = m.process();
    System.out.println(g);
    m = new WheatMill();
    g = m.process();
    System.out.println(g);
  }
} /* Output:
Grain
Wheat
*///:~
```

Różnica pomiędzy Javą SE5 a poprzednimi jej wersjami polega na tym, że wcześniejsze wersje wymuszały na przesłoniętej wersji `process()` zwracanie obiektu typu `Grain` zamiast `Wheat`, mimo że typ `Wheat` jest pochodną `Grain` i jako taki jest pełnoprawnym typem zwracanym. Kowariancja typów zwracanych pozwala na zwracanie bardziej specjalizowanego typu `Wheat`.

Projektowanie z użyciem dziedziczenia

Po odkryciu polimorfizmu łatwo jest pomyśleć, że ponieważ jest on tak sprytnym narzędziem, należy stosować dziedziczenie wszędzie. Może to mieć jednak negatywny wpływ na projekty. W istocie wybieranie dziedziczenia jako pierwszego sposobu w sytuacji, gdy wykorzystujemy istniejącą klasę do stworzenia nowej, powoduje, że sprawy niepotrzebnie się komplikują.

Lepszym wyjściem jest wybór nieoczywisty — wybieranie jako pierwszego przybliżenia kompozycji. Kompozycja nie wymusza budowania projektu jako hierarchii dziedziczenia. Jest także bardziej elastyczna, ponieważ możliwa jest dynamiczna zmiana typów (a przez to zachowania) składowych klasy, podczas gdy w wypadku dziedziczenia dokładny typ musi być znany na etapie kompilacji. Ilustruje to poniższy przykład:

```
//: polymorphism/Transmogrify.java
// Dynamiczna zmiana zachowania obiektu
// za pomocą kompozycji (wzorzec projektowy "Stan").
import static net.mindview.util.Print.*;

class Actor {
  public void act() {}
}

class HappyActor extends Actor {
  public void act() { print("HappyActor"); }
}

class SadActor extends Actor {
  public void act() { print("SadActor"); }
}

class Stage {
  private Actor actor = new HappyActor();
  public void change() { actor = new SadActor(); }
  public void performPlay() { actor.act(); }
}

public class Transmogrify {
  public static void main(String[] args) {
    Stage stage = new Stage();
    stage.performPlay();
    stage.change();
    stage.performPlay();
  }
} /* Output:
HappyActor
SadActor
*///:~
```

Obiekt typu `Stage` zawiera referencję do typu `Actor`, która jest inicjalizowana na obiekt klasy `HappyActor`. Oznacza to, że metoda `performPlay()` działa w pożądany sposób. Ponieważ jednak referencja może w czasie wykonania zostać związana z innym obiektem, możemy więc podstawić do a referencję do obiektu `SadActor`, zmieniając dzięki temu zachowanie metody `performPlay()`. Zyskujemy zatem dynamiczną elastyczność czasu

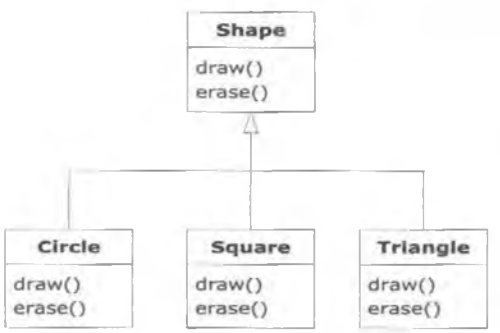
wykonania (rozwiązanie przedstawione tutaj odpowiada wzorcowi o nazwie „Stan" (ang. *State Pattern*, patrz książka *Thinking in Patterns (with Java)* dostępna w witrynie *www.MindView.net*)). Przeciwnie, nie możemy zdecydować się na inny sposób dziedziczenia w czasie wykonywania programu — musi to być jednoznacznie określone w chwili kompilacji.

Generalną wskazówką jest: „Używaj dziedziczenia do wyrażenia różnic w zachowaniu, pól zaś do wyrażenia zmian stanu". W powyższym przykładzie wykorzystuje się obie techniki: stworzone zostały dwie różne klasy pochodne do wyrażenia różnic w zachowaniu metody `act()`, natomiast obiekt `Stage` wykorzystuje kompozycję w celu umożliwienia zmiany swego stanu. W tym przypadku zmiana stanu powoduje zmianę w zachowaniu.

Ćwiczenie 16. Naśladując przykład z pliku *Transmogrify.java*, stwórz klasę `Starship` zawierającą referencję do typu `AlertStatus`, mogącą oznaczać jeden z trzech różnych stanów. Dodaj metody do zmiany stanów (3).

Substytucja kontra rozszerzanie

Zdaje się, że najbardziej porządnym sposobem tworzenia hierarchii dziedziczenia jest podejście „czyste", tj. ograniczenie do przesłaniania metod ustanowionych w klasie bazowej, tak jak to przedstawia poniższy diagram:

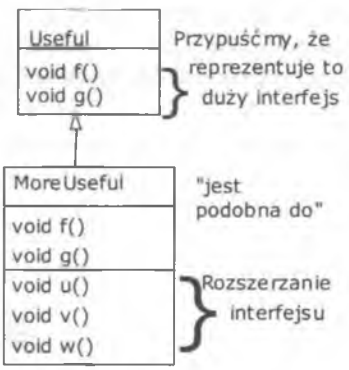


Można to określić jako czystą relację „bycia czymś", ponieważ to interfejs klasy określa, czym ona jest. Dziedziczenie gwarantuje, że klasy pochodne nic będą miały interfejsu mniejszego od klasy bazowej. Jeżeli będziemy postępować zgodnie ze strategią przedstawioną na powyższym diagramie, klasy pochodne nie będą także miały interfejsu *większego* od klasy bazowej.

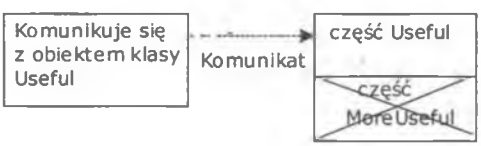
Sytuację taką możemy nazwać *czystą zastępowalnością*, ponieważ klasy pochodne mogą idealnie zastępować klasę bazową, a przy ich używaniu nie potrzebujemy nigdy żadnej dodatkowej informacji:

Tak więc klasa bazowa może otrzymywać wszystkie komunikaty, jakie możemy wysyłać do klas pochodnych, ponieważ mają one dokładnie taki sam interfejs. Wszystko, co musimy zrobić, to rzutować w górę z klasy pochodnej, a następnie nigdy więcej nie sprawdzać, z jakiego właściwie typu obiektem mamy do czynienia. Wszystkim zajmuje się polimorfizm.

Gdy spojrzymy na to w ten sposób, wydaje się, iż czysta relacja „bycia czymś" jest jedynym rozsądnym sposobem, każdy zaś inny projekt stanowi rezultat pokrętnego myślenia i z definicji jest zły. To także jest pułapka. Zaraz po tym, jak zaczniemy myśleć w ten sposób, odkryjemy, że rozszerzenie interfejsu (do którego, niestety, zachęca słowo kluczowe extends, czyli „rozszerza") stanowi idealne rozwiązanie jakiegoś problemu. Można to nazwać relacją „bycia podobnym do czegoś", ponieważ klasa pochodna jest *podobna* do klasy bazowej — ma ten sam bazowy interfejs — oprócz tego ma jednak dodatkowe właściwości, wymagające zaimplementowania dodatkowych metod:



Choć jest to również użyteczne i rozsądne (w zależności od sytuacji) rozwiązanie, ma jednak pewną wadę. Rozszerzona część interfejsu klasy pochodnej nie jest dostępna z klasy bazowej, zatem po wykonaniu rzutowania w górę nie możemy wywoływać nowych metod:



Jeżeli nie będziemy w tym przypadku rzutować w górę, ten problem nie będzie nas dotyczyć, często jednak jesteśmy zmuszeni odkryć na nowo dokładny typ danego obiektu, aby móc dostać się do wchodzących w skład rozszerzeń typu metod. Dalsza część opisuje, w jaki sposób to robić.

Rzutowanie w dół a identyfikacja typu w czasie wykonania

Ponieważ w czasie *rzutowania w górę* (podczas przechodzenia ku górze hierarchii dziedziczenia) tracimy informację o specyficznym typie obiektu, sensowne jest zatem jej odzyskanie z wykorzystaniem operacji *rzutowania w dół*. Między tymi operacjami zachodzi jednak bardzo istotna różnica: wiemy, że rzutowanie w górę jest zawsze bezpieczne,

bo klasa bazowa nie może mieć interfejsu większego od klasy pochodnej, zatem każdy komunikat odebrany poprzez interfejs klasy bazowej zostanie zaakceptowany. Przypadek rzutowania w dół jest inny: nie wiemy, czy np. dana figura jest okręgiem. Może zamiast tego być trójkątem, kwadratem lub czymś jeszcze innym.

W celu rozwiązania tego problemu musi istnieć jakiś sposób zapewnienia, że rzutowanie w dół jest poprawne, aby zapobiec przypadkowemu rzutowaniu na niewłaściwy typ i wysłaniu komunikatu, który nie zostanie zaakceptowany przez obiekt — byłoby to dosyć niebezpieczne.

W niektórych językach (jak w C++) musimy wykonać specjalną operację, by uzyskać bezpieczne rzutowanie w dół, w Javie jednak *każde rzutowanie* jest sprawdzane! A zatem, choć wydaje się, że przeprowadzamy po prostu zwyczajne rzutowanie, używając nazwy typu w nawiasach, to jednak w czasie wykonania rzutowanie takie jest sprawdzane w celu upewnienia się, że w rzeczywistości mamy do czynienia z typem, o jakim myślimy. Jeżeli tak nie jest, otrzymujemy wyjątek `ClassCastException`. To sprawdzanie typu w czasie wykonania nazywamy *identyfikacją typu w czasie wykonania* (ang. *run-time type information*, RTTI). Poniższy przykład pokazuje działanie RTTI:

```
//: polymorphism/RTTI.java
// Rzutowanie w dół i identyfkatja typu w czasie wykonania (RTTI).
// {ThrowsException}

class Useful {
  public void f() {}
  public void g() {}
}

class MoreUseful extends Useful {
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
}

public class RTTI {
  public static void main(String[] args) {
    Useful[] x = {
      new Useful(),
      new MoreUseful()
    };
    x[0].f();
    x[1].g();
    // W czasie kompilacji: brak szukanej metody w Useful:
    //! x[1].u();
    ((MoreUseful)x[1]).u(); // Rzutowanie w dół/RTTI
    ((MoreUseful)x[0]).u(); // Zgłoszenie wyjątku
  }
} ///:~
```

Podobnie jak na wcześniejszym diagramie, klasa `MoreUseful` rozszerza interfejs klasy `Useful`. Ponieważ dziedziczy po niej, zatem może być rzutowana na nią w górę. Rzutowanie takie ma miejsce podczas inicjalizacji zmiennej `x` w metodzie `main()`. Ponieważ

oba obiekty w tablicy są klasy `Useful`, zatem do obu można wysłać komunikaty `f()` i `g()`, przy próbie zaś wywołania metody `u()` (istniejącej jedynie w klasie `MoreUseful`) otrzymamy błąd kompilacji.

Jeśli chcielibyśmy dostać się do rozszerzonego interfejsu obiektu klasy `MoreUseful`, możemy spróbować rzutowania w dół. Uda nam się, jeżeli obiekt jest poprawnego typu. W przeciwnym razie otrzymamy wyjątek `ClassCastException`. Nie musimy pisać żadnego specjalnego kodu obsługi tego wyjątku, ponieważ oznacza on błąd programisty, który może zdarzyć się w dowolnym miejscu programu. Znacznik komentarza `{ThrowsException}` sygnalizuje stosowanemu przeze mnie systemowi kompilacji, aby spodziewał się zgłoszenia wyjątku przy uruchomieniu programu.

Mechanizm RTTI umożliwia znacznie więcej niż proste rzutowanie. Możemy na przykład sprawdzić typ obiektu, *zanim* spróbujemy rzutowania w dół. Różnym aspektom identyfikacji typu w czasie wykonania w Javie poświęcony jest cały rozdział „Informacje o typach".

Ćwiczenie 17. Bazując na hierarchii klas `Cycle` z ćwiczenia 1., dodaj metodę `balance()` do klas `Unicycle` i `Bicycle` (ale nie do `Tricycle`). Utwórz egzemplarze wszystkich trzech typów i przeprowadź rzutowanie w górę przy wstawianiu do tablicy elementów typu `Cycle`. Następnie spróbuj kolejno wywołać metodę `balance()` na rzecz wszystkich elementów tablicy i obserwuj efekty. W dalszej kolejności poddaj obiekty rzutowaniu w dół z wywołaniem `balance()` i obserwuj efekty (2).

Podsumowanie

Słowo polimorfizm oznacza „wielopostaciowość". W programowaniu zorientowanym obiektowo mamy do czynienia ze wspólnym interfejsem z klasy bazowej i różnymi wcieleniami tego interfejsu w postaci różnych wersji metod wiązanych dynamicznie.

W tym rozdziale zobaczyliśmy, że niemożliwe jest zrozumienie, a nawet stworzenie przykładu polimorfizmu bez użycia abstrakcji danych i dziedziczenia. Polimorfizm nie jest cechą, którą można zaobserwować w izolacji (tak jak np. instrukcję `switch`), lecz działającą jedynie w zespole, jako część złożonych relacji między klasami.

Aby w swoich programach efektywnie wykorzystywać polimorfizm — a przez to techniki obiektowe — musimy rozszerzyć sposób myślenia o programowaniu, tak aby objąć nie tylko składowe i komunikaty pojedynczej klasy, ale także podobieństwa między klasami i wzajemne relacje klas. Mimo że wymaga to wysiłku, nie jest on daremny — rezultatami są: szybsze tworzenie programów, lepsza organizacja kodu, rozszerzalność i łatwiejsza pielęgnacja kodu.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 9.

Interfejsy

Interfejsy i klasy abstrakcyjne to strukturalizowane środki oddzielenia interfejsu od implementacji.

Mechanizmy te nie są bynajmniej powszechne w językach programowania. Na przykład C++ obsługuje je jedynie pośrednio. Ich jawna obecność w Javie oznacza, że zostały uznane za dostatecznie ważne, aby były obsługiwane bezpośrednio przez słowa kluczowe języka.

Na pierwszy ogień pójdą *klasy abstrakcyjne*, jako postacie pośrednie pomiędzy najzwyklejszymi klasami a interfejsami. Choć w pierwszym impulsie chcielibyśmy utworzyć interfejs, klasa abstrakcyjna jest ważnym i niezbędnym narzędziem konstruowania klas, w których niektóre metody są celowo niezaimplementowane. Nie zawsze bowiem da się korzystać z czystych interfejsów.

Klasy i metody abstrakcyjne

We wszystkich przykładach z instrumentami (z poprzedniego rozdziału) metody klasy bazowej `Instrument` były zawsze „atrapami" — jeśli którakolwiek z nich zostałaby kiedyś wywołana, znaczyłoby to, że zrobiliśmy coś nie tak. Działo się tak dlatego, że przeznaczeniem klasy `Instrument` było stworzenie *wspólnego interfejsu* dla klas z niej wywiedzionych.

W owych przykładach jedynym powodem ustanowienia takiego wspólnego interfejsu jest możliwość różnorodnego jego wyrażania przez poszczególne podtypy. Interfejs narzucał podstawową postać, wspólną dla wszystkich klas pochodnych. Innymi słowy, klasę `Instrument` możemy nazwać *abstrakcyjną klasą bazową* lub krócej *klasą abstrakcyjną* (ang. *abstract class*).

Obiekty klasy abstrakcyjnej, takiej jak `Instrument`, niemal nigdy nie mają konkretnego znaczenia. Klasę abstrakcyjną tworzymy, gdy chcemy manipulować zestawem klas pochodnych poprzez wspólny interfejs. Klasa `Instrument` ma wyrażać tylko interfejs, a nie określoną implementację, zatem tworzenie obiektów klasy `Instrument` nie ma większego sensu i chcielibyśmy prawdopodobnie zabronić tego użytkownikowi. Można to osiągnąć, dodając do wszystkich metod tej klasy kod powodujący błędy, odsuwa to jednak uzyskanie informacji o błędzie aż do czasu wykonania programu, a poza tym wymaga ze strony

użytkownika pewnych wyczerpujących testów. Zawsze korzystniejsze jest wyłapywanie problemów już na etapie kompilacji.

Java dostarcza nam mechanizm zwany *metodą abstrakcyjną*[1]. Metoda taka jest niekompletna — zawiera tylko deklarację, nie posiada natomiast ciała. Składnia deklaracji takiej metody wygląda tak:

```
abstract void f();
```

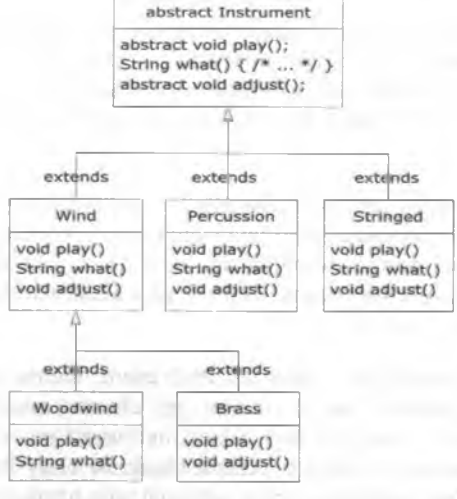
Klasa zawierająca metody abstrakcyjne nazywana jest *klasą abstrakcyjną*. Jeżeli klasa zawiera co najmniej jedną metodę abstrakcyjną, musi być zadeklarowana z użyciem kwalifikatora `abstract` — w przeciwnym razie kompilator zgłosi błąd.

Skoro klasa abstrakcyjna jest niekompletna, co zrobi kompilator, jeśli ktoś spróbuje stworzyć obiekt takiej klasy? Nie może bezpiecznie zrealizować takiego żądania, a zatem zwróci komunikat o błędzie. W ten sposób zapewnia czystość klasy abstrakcyjnej, dzięki czemu nie musimy się martwić o jej nieprawidłowe użycie.

Jeśli dziedziczymy po klasie abstrakcyjnej i chcemy tworzyć obiekty nowego typu, musimy dostarczyć definicje wszystkich metod abstrakcyjnych klasy bazowej. Jeśli tego nie zrobimy (a mamy taką możliwość), nowa klasa stanie się również abstrakcyjna i kompilator zmusi nas do użycia kwalifikatora `abstract` także w stosunku *do niej*.

Możliwe jest zadeklarowanie klasy jako abstrakcyjnej (z użyciem `abstract`) nawet wtedy, gdy nie zawiera ona żadnych metod abstrakcyjnych. Jest to przydatne w sytuacji, gdy chcemy zapobiec tworzeniu instancji naszej klasy, mimo że nie ma sensu deklarowanie którejkolwiek z jej metod jako abstrakcyjnej.

Klasa `Instrument` z poprzedniego rozdziału może być z łatwością zmieniona w klasę abstrakcyjną. Tylko niektóre z jej metod będą abstrakcyjne, jako że uczynienie abstrakcyjną klasy nie obliguje nas do zrobienia tego samego z wszystkimi jej metodami. Oto jak będzie to teraz wyglądać:



[1] Uwaga dla programistów C++: jest to rozwiązanie analogiczne do występujących w C++ *funkcji czysto wirtualnych* (ang. *pure virtual functions*).

A oto przykład z orkiestrą zmodyfikowany tak, aby używał abstrakcyjnych klas i metod:

```
// : interfaces/music4/Music4.java
// Abstrakcyjne klasy i metody.
package interfaces.music4;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

abstract class Instrument {
  private int i; // Pamięć przydzielana osobno dla każdego egzemplarza
  public abstract void play(Note n);
  public String what() { return "Instrument"; }
  public abstract void adjust();
}

class Wind extends Instrument {
  public void play(Note n) {
    print("Wind.play() " + n);
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion extends Instrument {
  public void play(Note n) {
    print("Percussion.play() " + n);
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed extends Instrument {
  public void play(Note n) {
    print("Stringed.play() " + n);
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play(Note n) {
    print("Brass.play() " + n);
  }
  public void adjust() { print("Brass.adjust()"); }
}

class Woodwind extends Wind {
  public void play(Note n) {
    print("Woodwind.play() " + n);
  }
  public String what() { return "Woodwind"; }
}

public class Music4 {
  // Metoda nie konkretyzuje typu, więc można z nią
  // stosować nowe typy dodawane do systemu:
  static void tune(Instrument i) {
    // ...
```

```
    i.play(Note.MIDDLE_C);
  }
  static void tuneAll(Instrument[] e) {
    for(Instrument i : e)
      tune(i);
  }
  public static void main(String[] args) {
    // Rzutowanie w górę w ramach wstawiania do tablicy:
    Instrument[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind()
    };
    tuneAll(orchestra);
  }
} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:~
```

Widzimy, że nie ma tu zmian, poza tymi w klasie bazowej.

Stosowanie kwalifikatora `abstract` jest pożyteczne, ponieważ jawnie określa on abstrakcyjność klasy, przez co informuje zarówno użytkownika, jak i kompilator, w jaki sposób chcielibyśmy, aby klasa była używana. Klasy abstrakcyjne są również przydatnymi narzędziami refaktoryzacji kodu, bo pozwalają na łatwe przenoszenie metod w górę hierarchii dziedziczenia.

Ćwiczenie 1. Zmodyfikuj ćwiczenie 9. z poprzedniego rozdziału tak, aby klasa `Gryzon` była abstrakcyjna. Abstrakcyjne powinny być też wszelkie metody klasy `Gryzon`, dla których jest to możliwe (1).

Ćwiczenie 2. Utwórz klasę abstrakcyjną bez metod abstrakcyjnych; sprawdź, czy możesz utworzyć jakiekolwiek egzemplarze takiej klasy (1).

Ćwiczenie 3. Utwórz klasę bazową z abstrakcyjną metodą `print()`, przesłoniętą w klasach pochodnych. Wersja przesłonięta metody ma wypisywać wartość składowej `int`, zdefiniowanej w klasie pochodnej. W miejscu definiowania zmiennej nadaj jej wartość niezerową. W konstruktorze klasy bazowej wywołaj metodę. W metodzie `main()` utwórz obiekt klasy pochodnej, a potem wywołaj na jego rzecz metodę `print()`. Wyjaśnij zaobserwowane efekty (2).

Ćwiczenie 4. Utwórz klasę abstrakcyjną bez żadnych metod. Wyprowadź z niej klasę pochodną i uzupełnij ją o metodę. Utwórz metodę statyczną, która przyjmuje referencję klasy bazowej, rzutuje ją w dół na typ klasy pochodnej i wywołuje metodę klasy pochodnej. Zademonstruj działanie całości w metodzie `main()`, a potem uzupełnij deklarację metody w klasie bazowej słowem `abstract`, eliminując tym samym potrzebę rzutowania (3).

Interfejsy

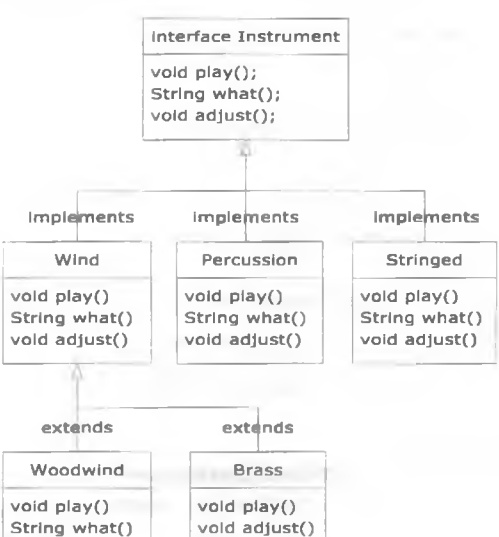
Słowo kluczowe `interface` to duży postęp, jeśli chodzi o koncepcję abstrakcyjności klas. Słowo kluczowe `abstract` pozwala na tworzenie w klasie jednej czy wielu metod niezdefiniowanych, a więc ustanowienie interfejsu bez udostępnienia odpowiadającej mu implementacji. Implementację mają dostarczyć klasy dziedziczące. Z kolei słowo kluczowe `interface` generuje zupełnie abstrakcyjną klasę bazową, która jest całkowicie pozbawiona implementacji. Klasa taka pozwala swemu twórcy ustanowić nazwy metod, listy argumentów, typy zwracane — jednak bez ciał metod. Interfejs ustanawia więc formę bez jakiejkolwiek treści.

Interfejs informuje: „Oto jak będą wyglądać wszystkie klasy *implementujące* mnie". Tak więc kod używający danego interfejsu wie, jakie metody mogą być wywołane dla tego interfejsu, i to wszystko. Interfejs ustanawia *protokół* pomiędzy klasami (w niektórych językach zorientowanych obiektowo funkcję tę pełni słowo kluczowe *protocol*).

Ale interfejs to nie tylko klasa bazowa o najwyższej możliwej abstrakcyjności; interfejs pozwala na realizację odmiany „dziedziczenia wielobazowego", czyli tworzenie klas, które dadzą się rzutować w górę na więcej niż jeden typ bazowy.

Aby stworzyć interfejs, musimy zastosować słowo kluczowe `interface` zamiast `class`. Tak jak w przypadku klasy możemy przed nim dodać słowo `public` (jednak tylko wtedy, gdy interfejs jest zadeklarowany w pliku o takiej samej nazwie jak jego nazwa) lub pozostawić dostęp pakietowy, by można było z niego korzystać jedynie z wnętrza tego samego pakietu. Interfejs może również zawierać pola, jednak są one automatycznie traktowane jako zadeklarowane z użyciem `static` i `final`. Interfejs dostarcza więc formę bez implementacji.

Aby stworzyć klasę realizującą dany interfejs (lub grupę interfejsów), używamy słowa kluczowego `implements`. Mówimy wtedy: „Interfejs określa, jak to wygląda, ale teraz powiemy, jak to *działa*". Poza tymi różnicami realizacja interfejsu wygląda tak samo jak dziedziczenie. Ilustruje to diagram z instrumentami:



Klasy `Woodwind` oraz `Brass` pokazują, że po zaimplementowaniu interfejsu powstaje normalna klasa, którą można rozszerzać w standardowy sposób.

Mimo że mamy możliwość jawnego określenia deklaracji w interfejsie jako publicznych, są one publiczne nawet wtedy, gdy tego nie zrobimy. A zatem przy implementowaniu interfejsu jego metody muszą zostać zdefiniowane jako publiczne. W przeciwnym razie przyjmuje się domyślny dostęp „przyjazny" („pakietowy"), co oznacza, że zredukowalibyśmy podczas dziedziczenia dostępność metod, a na to kompilator Javy nie pozwoli.

Możemy to zaobserwować w zmodyfikowanej wersji przykładu z instrumentami. Zauważmy, że w interfejsie wszystkie metody są tylko zadeklarowane — kompilator pozwala tylko na to. Dodatkowo wszystkie metody interfejsu `Instrument` zostały zadeklarowane z użyciem `public`, mimo że byłyby publiczne nawet bez tego:

```
// interfaces/music5/Music5.java
// Interfejsy.
package interfaces.music5;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

interface Instrument {
  // Stała czasu kompilacji:
  int VALUE = 5; // statyczna i finalna
  // Interfejs nie może posiadać definicji metod:
  void play(Note n); // Automatycznie publiczna
  void adjust();
}

class Wind implements Instrument {
  public void play(Note n) {
    print(this + ".play() " + n);
  }
  public String toString() { return "Wind"; }
  public void adjust() { print(this + ".adjust()"); }
}

class Percussion implements Instrument {
  public void play(Note n) {
    print(this + ".play() " + n);
  }
  public String toString() { return "Percussion"; }
  public void adjust() { print(this + ".adjust()"); }
}

class Stringed implements Instrument {
  public void play(Note n) {
    print(this + ".play() " + n);
  }
  public String toString() { return "Stringed"; }
  public void adjust() { print(this + ".adjust()"); }
}

class Brass extends Wind {
  public String toString() { return "Brass"; }
}
```

```
class Woodwind extends Wind {
  public String toString() { return "Woodwind"; }
}

public class Music5 {
  // Metoda nie konkretyzuje typu, więc można z nią
  // stosować nowe typy dodawane do systemu:
  static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  static void tuneAll(Instrument[] e) {
    for(Instrument i : e)
      tune(i);
  }
  public static void main(String[] args) {
    // Rzutowanie w górę w ramach wstawiania do tablicy:
    Instrument[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind()
    };
    tuneAll(orchestra);
  }
} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:~
```

Kod w tej wersji przykładu doczekał się zmiany dodatkowej: metoda `what()` została zamieniona na `toString()`, ponieważ i tak występowała w roli typowej dla `toString()`. Metoda `toString()` jest częścią klasy `Object`, więc nie musi występować jawnie w interfejsie.

Reszta kodu pozostała bez zmian. Nie jest istotne, czy rzutujemy w górę na „zwykłą" klasę `Instrument`, abstrakcyjną klasę `Instrument` czy interfejs o tej samej nazwie. Działanie jest takie samo. Można zaobserwować, że w metodzie `tune()` nie ma właściwie niczego, co pozwalałoby wywnioskować, czy `Instrument` jest klasą „zwykłą", abstrakcyjną czy też interfejsem.

Ćwiczenie 5. Utwórz interfejs zawierający trzy metody i zawrzyj go we własnym pakiecie. Zaimplementuj interfejs w innym pakiecie (2).

Ćwiczenie 6. Dowiedź, że wszystkie metody interfejsu są automatycznie metodami publicznymi (2).

Ćwiczenie 7. Zmień ćwiczenie 9. z rozdziału „Polimorfizm", czyniąc klasę `Gryzon` interfejsem (1).

Ćwiczenie 8. W pliku *polymorphizm/Sandwich.java* stwórz interfejs o nazwie `FastFood` (z odpowiednimi metodami) i zmień klasę `Sandwich` tak, aby implementowała także `FastFood` (2).

Ćwiczenie 9. Przebuduj plik *Music5.java*, przesuwając wspólne metody klas `Wind`, `Percussion` i `Stringed` do klasy abstrakcyjnej (3).

Ćwiczenie 10. Zmodyfikuj plik *Music5.java* poprzez dodanie do niego interfejsu `Playable`. Usuń deklarację `play()` z klasy `Instrument`. Dodaj `Playable` do klas pochodnych poprzez umieszczenie go na liście po `implements`. Zmień metodę `tune()` tak, aby pobierała argument typu `Playable` zamiast `Instrument` (3).

Rozdzielenie zupełne

Kiedy metoda operuje na klasie, a nie na interfejsie, jesteśmy ograniczeni do używania tej klasy albo jej podklas. Gdybyśmy zechcieli zaaplikować metodę do klasy spoza hierarchii, obeszlibyśmy się smakiem. Interfejs znacząco rozluźnia ten rodzaj związku. W efekcie otrzymujemy kod o większej przydatności do wielokrotnego użycia.

Załóżmy na przykład, że posiadamy klasę `Processor`, której metody `name()` i `process()` przyjmują dane wejściowe, modyfikują je i generują dane wyjściowe. Klasa bazowa jest rozszerzana na różne pochodne typy `Processor`. W takim przypadku podtypy `Processor` modyfikują obiekty `String` (zauważ, że kowariancja dotyczy typów wartości zwracanych, ale nie typów argumentów):

```
//: interfaces/classprocessor/Apply.java
package interfaces.classprocessor;
import java.util.*;
import static net.mindview.util.Print.*;

class Processor {
  public String name() {
    return getClass().getSimpleName();
  }
  Object process(Object input) { return input; }
}

class Upcase extends Processor {
  String process(Object input) { // Kowariancja typu zwracanego
    return ((String)input).toUpperCase();
  }
}

class Downcase extends Processor {
  String process(Object input) {
    return ((String)input).toLowerCase();
  }
}

class Splitter extends Processor {
  String process(Object input) {
    // Metoda split() dzieli ciąg na podciągi:
    return Arrays.toString(((String)input).split(" "));
  }
}
```

```
public class Apply {
  public static void process(Processor p, Object s) {
    print("Używam procesora " + p.name());
    print(p.process(s));
  }
  public static String s =
    "Idzie Grześ przez wieś, worek piasku niesie";
  public static void main(String[] args) {
    process(new Upcase(), s);
    process(new Downcase(), s);
    process(new Splitter(), s);
  }
} /* Output:
Używam procesora Upcase
IDZIE GRZEŚ PRZEZ WIEŚ, WOREK PIASKU NIESIE
Używam procesora Downcase
idzie grześ przez wieś, worek piasku niesie
Używam procesora Splitter
[Idzie, Grześ, przez, wieś, worek, piasku, niesie]
*///:~
```

Metoda `Apply.process()` przyjmuje na wejście obiekt dowolnego typu `Processor` i wykorzystuje go do przetworzenia obiektu `Object`, wypisując wynik na wyjściu programu. Utworzenie metody, której zachowanie zależy od obiektu argumentu, to realizacja wzorca projektowego *Strategy*. Metoda definiuje niezmienną część implementacji algorytmu do wykonania, a część różnicowana to już kwestia przyjęcia odpowiedniej strategii. Strategię wyraża tu obiekt `Processor`; w metodzie `main()` widzimy zastosowanie różnych strategii przetwarzania obiektu `String s`.

Metoda `split()` wchodzi w skład klasy `String`. Operuje na obiekcie, na rzecz którego została wywołana, a jej wartością zwracaną jest tablica `String[]` z podciągami powstałymi w wyniku podziału przekazanego ciągu według separatora podanego argumentem wywołania. To wygodny sposób tworzenia tablicy obiektów `String`.

Załóżmy teraz, że otrzymaliśmy do dyspozycji zestaw dodatkowych filtrów nadających się do zastosowania w metodzie `Apply.process()`:

```
//: interfaces/filters/Waveform.java
package interfaces.filters;

public class Waveform {
  private static long counter;
  private final long id = counter++;
  public String toString() { return "Waveform " + id; }
} ///:~

//: interfaces/filters/Filter.java
package interfaces.filters;

public class Filter {
  public String name() {
    return getClass().getSimpleName();
  }
  public Waveform process(Waveform input) { return input; }
} ///:~
```

```
// interfaces/filters/LowPass.java
package interfaces.filters;

public class LowPass extends Filter {
  double cutoff;
  public LowPass(double cutoff) { this.cutoff = cutoff; }
  public Waveform process(Waveform input) {
    return input; // Pozorowanie przetwarzania
  }
} ///:~

// interfaces/filters/HighPass.java
package interfaces.filters;

public class HighPass extends Filter {
  double cutoff;
  public HighPass(double cutoff) { this.cutoff = cutoff; }
  public Waveform process(Waveform input) { return input; }
} ///:~

// interfaces/filters/BandPass.java
package interfaces.filters;

public class BandPass extends Filter {
  double lowCutoff, highCutoff;
  public BandPass(double lowCut, double highCut) {
    lowCutoff = lowCut;
    highCutoff = highCut;
  }
  public Waveform process(Waveform input) { return input; }
} ///:~
```

`Filter` ma ten sam interfejs co `Processor`, ale ponieważ nie dziedziczy po klasie `Processor` (bo twórca klasy `Filter` nie miał pojęcia, że taka klasa istnieje), nie możemy użyć filtra `Filter` w metodzie `Apply.process()` — szkoda, bo pasowałby tam idealnie. Powiązanie pomiędzy `Apply.process()` a `Processor` jest najwyraźniej silniejsze niż powinno, co uniemożliwia metodzie `Apply.process()` wykorzystanie w innych, ale podobnych kontekstach. Zauważ też, że i wejścia, i wyjścia to obiekty `Waveform`.

Gdyby jednak `Processor` był interfejsem, krępujące więzy zostałyby poluzowane na tyle, że stosującą ten interfejs metodę `Apply.process()` można by wykorzystać również inaczej. Oto odpowiednio zmodyfikowane wersje `Processor` i `Apply`:

```
// interfaces/interfaceprocessor/Processor.java
package interfaces.interfaceprocessor;

public interface Processor {
  String name();
  Object process(Object input);
} ///:~

// interfaces/interfaceprocessor/Apply.java
package interfaces.interfaceprocessor;
import static net.mindview.util.Print.*;

public class Apply {
```

```
  public static void process(Processor p, Object s) {
    print("Używam procesora " + p.name());
    print(p.process(s));
  }
} ///:~
```

Pierwsza możliwość ponownego wykorzystania kodu polega na tym, że programista-klient może napisać własne klasy odpowiadające interfejsowi, jak tu:

```
//: interfaces/interfaceprocessor/StringProcessor.java
package interfaces.interfaceprocessor;
import java.util.*;

public abstract class StringProcessor implements Processor{
  public String name() {
    return getClass().getSimpleName();
  }
  public abstract String process(Object input);
  public static String s =
    "Jeśli ona waży tyle, co kaczka, to znaczy że jest z drewna";
  public static void main(String[] args) {
    Apply.process(new Upcase(), s);
    Apply.process(new Downcase(), s);
    Apply.process(new Splitter(), s);
  }
}

class Upcase extends StringProcessor {
  public String process(Object input) { // Kowariancja typu zwracanego
    return ((String)input).toUpperCase();
  }
}

class Downcase extends StringProcessor {
  public String process(Object input) {
    return ((String)input).toLowerCase();
  }
}

class Splitter extends StringProcessor {
  public String process(Object input) {
    return Arrays.toString(((String)input).split(" "));
  }
} /* Output:
Używam procesora Upcase
JEŚLI ONA WAŻY TYLE, CO KACZKA, TO ZNACZY, ŻE JEST Z DREWNA
Używam procesora Downcase
jeśli ona waży tyle, co kaczka, to znaczy, że jest z drewna
Używam procesora Splitter
[Jeśli, ona, waży, tyle,, co, kaczka,, to, znaczy, że, jest, z, drewna]
*///:~
```

Nie zawsze jednak można pozwolić sobie na modyfikowanie klas, których chce się używać. Owe elektroniczne filtry otrzymaliśmy gotowe, z zewnątrz. W takich przypadkach uciekamy się do wzorca projektowego *Adapter*. Polega to na napisaniu kodu, który przyjmowałby zastany interfejs i generował interfejs pożądany, jak tu:

```
//: interfaces/interfaceprocessor/FilterProcessor.java
package interfaces.interfaceprocessor;
import interfaces.filters.*;

class FilterAdapter implements Processor {
  Filter filter;
  public FilterAdapter(Filter filter) {
    this.filter = filter;
  }
  public String name() { return filter.name(); }
  public Waveform process(Object input) {
    return filter.process((Waveform)input);
  }
}

public class FilterProcessor {
  public static void main(String[] args) {
    Waveform w = new Waveform();
    Apply.process(new FilterAdapter(new LowPass(1.0)), w);
    Apply.process(new FilterAdapter(new HighPass(2.0)), w);
    Apply.process(
      new FilterAdapter(new BandPass(3.0, 4.0)), w);
  }
} /* Output:
Używam procesora LowPass
Waveform 0
Używam procesora HighPass
Waveform 0
Używam procesora BandPass
Waveform 0
*///:~
```

W tym wcieleniu Adaptera konstruktor klasy `FilterAdapter` przyjmuje narzucony interfejs — `Filter` — i generuje obiekt dysponujący potrzebnym nam interfejsem `Processor`. W klasie `FilterAdapter` widać też zastosowanie techniki delegacji.

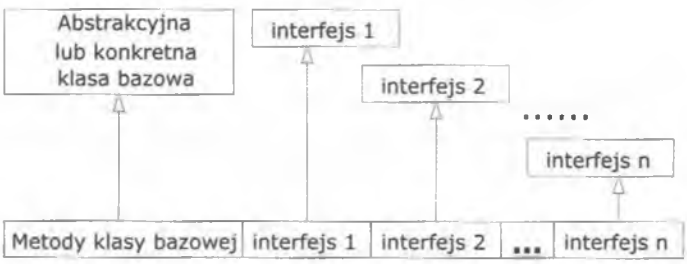
Rozluźnianie związku pomiędzy interfejsem a implementacją pozwala interfejsowi na stosowanie z wieloma różnymi implementacjami, co zwiększa przydatność kodu do ponownego użycia.

Ćwiczenie 11. Utwórz klasę z metodą przyjmującą argument typu `String` i zwracającą takiż obiekt, z ciągiem powstałym przez zamianę par znaków w ciągu argumentu. Zaadaptuj klasę tak, aby działała z metodą `interfaceprocessor.Apply.process()` (4).

„Dziedziczenie wielobazowe" w Javie

Ponieważ interfejs nie posiada żadnej implementacji — nie jest z nim związana żadna pamięć — nic nie stoi na przeszkodzie, aby połączyć wiele interfejsów. Możliwość taka jest cenna, ponieważ w pewnych sytuacjach chcielibyśmy powiedzieć: „x jest rodzaju a, b i c". W C++ akt łączenia interfejsów wielu klas nazywany jest *wielokrotnym dziedzi-*

czeniem i stwarza wiele niebezpieczeństw, ponieważ każda klasa może mieć implementację. W Javie możemy dokonać czegoś podobnego, jednak tylko jedna z klas może mieć implementację, a zatem problemy występujące w C++ podczas łączenia wielu interfejsów w Javie nie występują:



Klasa pochodna nie musi mieć abstrakcyjnej lub „konkretnej" (tj. bez metod abstrakcyjnych) klasy bazowej. Jeżeli jednak *już* dziedziczymy po klasie, a nie interfejsie, wtedy może to być tylko jedna klasa. Wszystkie inne elementy bazowe muszą być interfejsami. Wszystkie nazwy interfejsów umieszczamy po słowie kluczowym `implements` i rozdzielamy przecinkami. Możemy ich mieć tyle, ile chcemy; możemy też rzutować klasę w górę na dowolny z nich, bo każdy interfejs stanowi niezależny typ. Następujący przykład pokazuje połączenie klasy konkretnej z kilkoma interfejsami w celu stworzenia nowej klasy:

```
//: interfaces/Adventure.java
// Wiele interfejsów.

interface CanFight {
  void fight();
}

interface CanSwim {
  void swim();
}

interface CanFly {
  void fly();
}

class ActionCharacter {
  public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
  public void swim() {}
  public void fly() {}
}

public class Adventure {
  public static void t(CanFight x) { x.fight(); }
  public static void u(CanSwim x) { x.swim(); }
  public static void v(CanFly x) { x.fly(); }
  public static void w(ActionCharacter x) { x.fight(); }
```

```
  public static void main(String[] args) {
    Hero h = new Hero();
    t(h); // Obiekt Hero jako CanFight
    u(h); // Obiekt Hero jako CanSwim
    v(h); // Obiekt Hero jako CanFly
    w(h); // Obiekt Hero jako ActionCharacter
  }
} ///:~
```

Widzimy, że klasa `Hero` łączy konkretną klasę `ActionCharacter` z interfejsami: `CanFight`, `CanSwim`, `CanFly`. Dokonując takiego łączenia, musimy najpierw wymienić klasę konkretną, a po niej interfejsy (w przeciwnym razie kompilator zaprotestuje).

Sygnatura metody `fight()` jest taka sama w interfejsie `CanFight` i klasie `ActionCharacter` oraz że metoda o takiej nazwie *nie* jest dostarczona w definicji klasy `Hero`. Regułą jest, że po interfejsach można dziedziczyć — jednak otrzymujemy wtedy kolejny interfejs. Jeżeli chcemy stworzyć obiekt powstałego typu, musi być on klasą z dostarczonymi wszystkimi definicjami. Chociaż klasa `Hero` nie dostarcza definicji metody `fight()`, to jednak otrzymuje taką definicję, dziedzicząc po `ActionCharacter`, a zatem tworzenie obiektów typu `Hero` jest możliwe.

W klasie `Adventure` występują cztery metody pobierające jako argumenty różne interfejsy oraz klasę konkretną. Po stworzeniu obiektu `Hero` może on być przekazywany do wszystkich tych metod, co z kolei oznacza, że możliwe jest rzutowanie na każdy z interfejsów. Dzieje się to bez żadnych problemów i bez dodatkowego wysiłku ze strony programisty dzięki sposobowi, w jaki zostały w Javie zaprojektowane interfejsy.

Musimy pamiętać, że jednym z głównych powodów wprowadzenia interfejsów jest (pokazana w powyższym przykładzie) możliwość rzutowania na kilka typów bazowych. Drugi powód jest natomiast taki sam, jak w przypadku stosowania klas abstrakcyjnych: należy uniemożliwić programiście-klientowi tworzenie obiektów naszej klasy i zaznaczyć, że stanowi ona tylko interfejs.

Nasuwa się tu pytanie, kiedy powinniśmy stosować interfejsy, a kiedy klasy abstrakcyjne. Jeżeli możliwe jest stworzenie klasy bazowej bez definiowania żadnych metod lub zmiennych składowych, powinniśmy zawsze wybierać interfejsy, a nie klasy abstrakcyjne. W zasadzie, jeżeli wiemy, że coś będzie jedynie klasą bazową, naszym pierwszym wyborem powinno być uczynienie tego interfejsem (do tego tematu wrócimy w podsumowaniu rozdziału).

Ćwiczenie 12. W pliku *Adventure.java* dodaj interfejs o nazwie `CanClimb`; powinien naśladować pozostałe stosowane tam interfejsy (2).

Ćwiczenie 13. Utwórz interfejs i wyprowadź z niego dwa nowe interfejsy. Z owych dwóch wyprowadź (przez dziedziczenie wielobazowe) jeszcze jeden interfejs[2] (2).

[2] Będzie to ilustracja „problemu rombu" pojawiającego się przy dziedziczeniu wielobazowym w C++.

Rozszerzanie interfejsu poprzez dziedziczenie

Używając dziedziczenia, możemy z łatwością dodać nowe metody do interfejsu, a także połączyć kilka interfejsów w jeden. W obu przypadkach otrzymujemy nowy interfejs, jak pokazuje przykład:

```
//: interfaces/HorrorShow.java
// Rozszerzanie interfejsu przez dziedziczenie.

interface Monster {
  void menace();
}

interface DangerousMonster extends Monster {
  void destroy();
}

interface Lethal {
  void kill();
}

class DragonZilla implements DangerousMonster {
  public void menace() {}
  public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
  void drinkBlood();
}

class VeryBadVampire implements Vampire {
  public void menace() {}
  public void destroy() {}
  public void kill() {}
  public void drinkBlood() {}
}

public class HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  static void w(Lethal l) { l.kill(); }
  public static void main(String[] args) {
    DangerousMonster barney = new DragonZilla();
    u(barney);
    v(barney);
    Vampire vlad = new VeryBadVampire();
    u(vlad);
    v(vlad);
    w(vlad);
  }
} ///:~
```

`DangerousMonster` jest prostym rozszerzeniem `Monster`, które daje nowy interfejs. Ten nowy interfejs jest implementowany przez klasę `DragonZilla`.

Składnia zastosowana w interfejsie `Vampire` działa *jedynie* przy dziedziczeniu po interfejsach. Normalnie możemy stosować `extends` jedynie z pojedynczą klasą, jednak słowo `extends` może odnosić się do wielu interfejsów bazowych składających się na jeden interfejs wynikowy. Jak widać, nazwy interfejsów są po prostu oddzielane przecinkami.

Ćwiczenie 14. Stwórz trzy interfejsy, każdy z dwoma metodami. Wyprowadź z nich przez dziedziczenie nowy interfejs, dodając przy okazji jeszcze jedną metodę. Stwórz klasę implementującą ten nowy interfejs i dziedziczącą po jakiejś klasie konkretnej. Napisz teraz cztery metody, z których każda pobiera jako argument inny z czterech interfejsów. W metodzie `main()` stwórz obiekt Twojej klasy i przekaż go każdej z czterech metod (2).

Ćwiczenie 15. Zmodyfikuj ćwiczenie 14. poprzez stworzenie klasy abstrakcyjnej i dziedziczenie po niej w klasie pochodnej (2).

Kolizje nazw podczas łączenia interfejsów

Implementacja wielu interfejsów przez jedną klasę stwarza małą pułapkę. W poprzednim przykładzie zarówno interfejs `CanFight`, jak i klasa `ActionCharacter` miały identyczną metodę `void fight()`. Nie stanowiło to problemu, ponieważ metody były w obu przypadkach identyczne. Co by się jednak stało, gdyby takie nie były? Oto przykład:

```
//: interfaces/InterfaceCollision.java
package interfaces;

interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
  public void f() {}
  public int f(int i) { return 1; } // przeciążona
}

class C3 extends C implements I2 {
  public int f(int i) { return 1; } // przeciążona
}

class C4 extends C implements I3 {
  // Identyczne, bez problemu:
  public int f() { return 1; }
}

// Metody różnią się jedynie typem wartości zwracanej:
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} ///:~
```

Problem pojawia się, ponieważ przesłanianie, implementacja i przeciążanie są tu wymieszane, a funkcje przeciążone nie mogą różnić się jedynie typem wartości zwracanej. Po usunięciu komentarza z dwóch ostatnich wierszy poinformują nas o tym komunikaty o błędach:

InterfaceCollision.java:23: f() in C cannot implement f() in I1; attempting to use incompatible return type

found: int

required: void

InterfaceCollision.java:24: interfaces I3 and I1 are incompatible; both define f(), but with different return type

Wykorzystywanie takich samych nazw metod w różnych interfejsach, przeznaczonych do łączenia ze sobą, powoduje zamieszanie i zmniejsza czytelność kodu. Należy tego unikać.

Adaptowanie do interfejsu

Jednym z uzasadnień dla istnienia i stosowania interfejsów jest umożliwienie rozmaitych implementacji danego interfejsu. W prostych przypadkach oznacza to metodę przyjmującą jakiś interfejs; to my musimy wtedy zaimplementować ten interfejs i przekazać implementujący go obiekt do metody.

Wynika z tego, że jednym z częstszych zastosowań interfejsów jest wcielanie wspomnianego już wzorca projektowego *Strategy*. Polega to na napisaniu metody realizującej pewne operacje, która przyjmuje pewien interfejs. Oznacza to wyrażenie: „możesz używać mojej metody z dowolnymi obiektami, byle te obiekty spełniały wymogi narzucane przez mój interfejs". W ten sposób metoda staje się bardziej uniwersalna i elastyczna, a tym samym zdatna do wielokrotnego wykorzystania.

Na przykład konstruktor klasy `Scanner` z Javy SE5 (o której powiemy sobie więcej w rozdziale „Ciągi znaków") wymaga przekazania implementacji interfejsu `Readable`. Taki argument jest ewenementem — nie ma go w żadnej innej metodzie biblioteki standardowej języka Java — został utworzony wyłącznie na potrzeby klasy `Scanner`, tak aby metody tej klasy nie były ograniczone do argumentów jakiejś innej konkretnej klasy. Dzięki temu klasa `Scanner` może pracować na rozmaitych typach. Aby obiekty swojej własnej klasy uzdatnić do przetwarzania klasą `Scanner`, wystarczy w swojej klasie zaimplementować interfejs `Readable`:

```
// : interfaces/RandomWords.java
// Implementacja interfejsu na potrzeby metody.
import java.nio.*;
import java.util.*;

public class RandomWords implements Readable {
  private static Random rand = new Random(47);
  private static final char[] capitals =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
  private static final char[] lowers =
    "abcdefghijklmnopqrstuvwxyz".toCharArray();
  private static final char[] vowels =
    "aeiou".toCharArray();
  private int count;
  public RandomWords(int count) { this.count = count; }
```

```
    public int read(CharBuffer cb) {
      if(count-- == 0)
        return -1; // Sygnalizuje koniec danych wejściowych
      cb.append(capitals[rand.nextInt(capitals.length)]);
      for(int i = 0; i < 4; i++) {
        cb.append(vowels[rand.nextInt(vowels.length)]);
        cb.append(lowers[rand.nextInt(lowers.length)]);
      }
      cb.append(" ");
      return 10; // Liczba dołączonych znaków
    }
    public static void main(String[] args) {
      Scanner s = new Scanner(new RandomWords(10));
      while(s.hasNext())
        System.out.println(s.next());
    }
} /* Output:
Yazeruyac
Fowenucor
Goeazimom
Raeuuacio
Nuoadesiw
Hageaikux
Ruqicibui
Numasetih
Kuuuuozog
Waqizeyoy
*///:~
```

Interfejs `Readable` wymaga jedynie implementacji metody `read()`. Wewnątrz `read()` dołączamy dane do argumentu `CharBuffer` (można to robić na kilka sposobów — szczegółów szukaj w dokumentacji klasy `CharBuffer`) albo zwracamy wartość -1, co oznacza wyczerpanie danych.

Załóżmy, że mamy klasę, która nie implementuje jeszcze interfejsu `Readable`. Jak przysposobić ją do współpracy ze wspomnianą metodą klasy `Scanner`? Oto przykład klasy, która generuje losowe wartości zmiennoprzecinkowe:

```
//: interfaces/RandomDoubles.java
import java.util.*;

public class RandomDoubles {
  private static Random rand = new Random(47);
  public double next() { return rand.nextDouble(); }
  public static void main(String[] args) {
    RandomDoubles rd = new RandomDoubles();
    for(int i = 0; i < 7; i ++)
      System.out.print(rd.next() + " ");
  }
} /* Output:
0.7271157860730044 0.5309454508634242 0.16020656493302599 0.18847866977771732
0.5166020801268457 0.2678662084200585 0.2613610344283964
*///:~
```

Ponownie uciekniemy się do wzorca Adapter, ale tym razem adaptacja będzie się odbywać przez dziedziczenie po interfejsie `Readable` (i jego implementację). Przy użyciu nibywielodziedziczenia wyrażanego przez słowo kluczowe `implements` tworzymy nową klasę, która jest zarówno typu `RandomDoubles`, jak i typu `Readable`:

```
//: interfaces/AdaptedRandomDoubles.java
// Tworzenie adaptera przez dziedziczenie.
import java.nio.*;
import java.util.*;

public class AdaptedRandomDoubles extends RandomDoubles
implements Readable {
  private int count;
  public AdaptedRandomDoubles(int count) {
    this.count = count;
  }
  public int read(CharBuffer cb) {
    if(count-- == 0)
      return -1;
    String result = Double.toString(next()) + " ";
    cb.append(result);
    return result.length();
  }
  public static void main(String[] args) {
    Scanner s = new Scanner(new AdaptedRandomDoubles(7));
    s.useLocale(Locale.ENGLISH);[3]
    while(s.hasNextDouble())
      System.out.print(s.nextDouble() + " ");
  }
} /* Output:
0.7271157860730044 0.5309454508634242 0.16020656493302599 0.18847866977771732
0.5166020801268457 0.2678662084200585 0.2613610344283964
*///:~
```

W ten sposób można uzupełniać istniejące klasy o nowe interfejsy, więc wyodrębnienie w klasie metody, która wymaga przekazania interfejsu, to świetny sposób na umożliwienie adaptacji dowolnych klas do wymogów tejże metody. W tym kontekście mamy do czynienia z wyższością interfejsów nad klasami.

Ćwiczenie 16. Utwórz klasę generującą sekwencję znaków char. Zaadaptuj tę klasę do wymogów metody klasy Scanner (3).

Pola w interfejsach

Ponieważ każde pole umieszczone w interfejsie staje się automatycznie statyczne i finalne, interfejsy stanowią wygodne narzędzie grupowania stałych. Był to swego czasu (przed pojawieniem się Javy SE5) jedyny sposób uzyskania tworu przypominającego wyliczenia (enum) z C++. W kodzie dla tamtych wersji Javy widuje się więc niejednokrotnie coś takiego:

[3] Ustawienie schematu lokalizacji jest konieczne, bo według schematu domyślnego w polskojęzycznej wersji systemu operacyjnego znak przecinka dziesiętnego w ciągach reprezentujących liczby double to przecinek (,) a w schematach anglojęzycznych rolę tę pełni kropka). Niedopasowanie tych znaków powoduje, że obiekt Scanner (inicjalizowany z domyślnym schematem lokalizacji) nie rozpoznaje wartości generowanych przez AdaptedRandomDoubles jako liczb double — *przyp. tłum.*

```
// interfaces/Months.java
// Użycie interfejsów do grupowania stałych.
package interfaces;

public interface Months {
  int
    JANUARY = 1, FEBRUARY = 2, MARCH = 3,
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
    NOVEMBER = 11, DECEMBER = 12;
} ///:~
```

Zwróćmy uwagę na stosowany w Javie zwyczaj używania wyłącznie wielkich liter (i znaku podkreślenia do oddzielania słów) w nazwach pól typu `static final`, posiadających stałe wartości inicjalizujące. Pola interfejsu są automatycznie publiczne, zatem także tego nie trzeba specyfikować.

W Javie SE5 mamy do dyspozycji dalece elastyczniejsze słowo kluczowe `enum`, które znosi potrzebę stosowania interfejsów do grupowania stałych w zbiory wyliczeniowe. Jednak z pewnością niejednokrotnie natkniesz się na kod bazujący na starych rozwiązaniach (pełny opis generowania typów wyliczeniowych przy użyciu interfejsów w wydaniach poprzedzających Javę SE5 można znaleźć w poprzednich wydaniach książki, publikowanych w witrynie *www.MindView.net*). Wyliczenia zostaną osobno omówione w rozdziale „Typy wyliczeniowe".

Ćwiczenie 17. Wykaż, że wszelkie pola interfejsu są niejawnie statyczne i finalne (2).

Inicjalizacja pól interfejsów

Pola zdefiniowane w interfejsach nie mogą być „pustymi zmiennymi finalnymi", ale mogą być inicjalizowane przez wyrażenia nie będące stałymi, na przykład:

```
// interfaces/RandVals.java
// Inicjalizacja pól interfejsu
// wyrażeniem niestałym.
import java.util.*;

public interface RandVals {
  Random RAND = new Random(47);
  int RANDOM_INT = RAND.nextInt(10);
  long RANDOM_LONG = RAND.nextLong() * 10;
  float RANDOM_FLOAT = RAND.nextLong() * 10;
  double RANDOM_DOUBLE = RAND.nextDouble() * 10;
} ///:~
```

Ponieważ pola te są statyczne, są inicjalizowane podczas ładowania klasy, co ma miejsce przy pierwszym odwołaniu do któregoś z nich. Oto prosty test:

```
// interfaces/TestRandVals.java
import static net.mindview.util.Print.*;

public class TestRandVals {
  public static void main(String[] args) {
    print(RandVals.RANDOM_INT);
```

```
    print(RandVals.RANDOM_LONG);
    print(RandVals.RANDOM_FLOAT);
    print(RandVals.RANDOM_DOUBLE);
  }
} /* Output:
8
-32032247016559954
-8.5939291E18
5.779976127815049
*///:~
```

Pola takie są oczywiście przechowywane w obszarze pamięci statycznej, związanym z danym interfejsem.

Zagnieżdżanie interfejsów

Interfejsy mogą być zagnieżdżane w klasach i innych interfejsach[4]. Stwarza to wiele bardzo interesujących możliwości:

```
//: interfaces/nesting/NestingInterfaces.java
package interfaces.nesting;

class A {
  interface B {
    void f();
  }
  public class BImp implements B {
    public void f() {}
  }
  private class BImp2 implements B {
    public void f() {}
  }
  public interface C {
    void f();
  }
  class CImp implements C {
    public void f() {}
  }
  private class CImp2 implements C {
    public void f() {}
  }
  private interface D {
    void f();
  }
  private class DImp implements D {
    public void f() {}
  }
  public class DImp2 implements D {
    public void f() {}
  }
  public D getD() { return new DImp2(); }
  private D dRef;
```

[4] Dziękuję Martinowi Dannerowi za pytanie dotyczące tego zagadnienia podczas seminarium.

```
  public void receiveD(D d) {
    dRef = d;
    dRef.f();
  }
}

interface E {
  interface G {
    void f();
  }
  // Słowo "public" jest zbędne:
  public interface H {
    void f();
  }
  void g();
  // Nie można być prywatnym wewnątrz interfejsu:
  //! private interface I {}
}

public class NestingInterfaces {
  public class BImp implements A.B {
    public void f() {}
  }
  class CImp implements A.C {
    public void f() {}
  }
  // Nie można implementować interfejsu prywatnego
  // poza klasą definiującą interfejs:
  //! class DImp implements A.D {
  //!   public void f() {}
  //! }
  class EImp implements E {
    public void g() {}
  }
  class EGImp implements E.G {
    public void f() {}
  }
  class EImp2 implements E {
    public void g() {}
    class EG implements E.G {
      public void f() {}
    }
  }
  public static void main(String[] args) {
    A a = new A();
    // Brak dostępu do A.D:
    //! A.D ad = a.getD();
    // Nie zwraca niczego poza A.D:
    //! A.DImp2 di2 = a.getD();
    // Brak dostępu do składowej interfejsu:
    //! a.getD().f();
    // Tylko inny obiekt A może zrobić coś z getD():
    A a2 = new A();
    a2.receiveD(a.getD());
  }
} ///:~
```

Składnia zagnieżdżania interfejsów w klasach jest oczywista i — tak jak w przypadku interfejsów niezagnieżdżonych — mogą one mieć zakres dostępu publiczny lub pakietowy.

Aby było ciekawiej, interfejsy mogą być również prywatne lub chronione. Interfejs prywatny pokazano w przypadku `A.D` (w stosunku do zagnieżdżonych interfejsów i zagnieżdżonych klas stosowana jest ta sama składnia kwalifikacji). Jaki może być pożytek z prywatnego interfejsu? Można się domyślać, że może on być implementowany jedynie przez prywatną klasę zagnieżdżoną, taką jak `DImp`, jednakże `A.Dimp2` pokazuje, że klasą implementującą może być również klasa publiczna. `A.Dimp2` może być jednak używana jedynie jako ona sama — nie możemy nigdzie poza klasą `A` wspominać, że implementuje ona prywatny interfejs. A zatem implementowanie prywatnego interfejsu jest jedynie sposobem wymuszenia definicji wszystkich jego metod, nie pociąga za sobą dodawania żadnej informacji o typie (tj. nie umożliwia żadnego rzutowania w górę).

Metoda `getD()` prowokuje kolejne pytanie dotyczące interfejsów prywatnych: jest ona metodą publiczną zwracającą taki interfejs. Co możemy zrobić z wartością zwróconą przez tę metodę? W `main()` widzimy kilka prób jej użycia — wszystkie niepoprawne. Jedyną rzeczą, która działa, jest przekazanie takiej zwróconej wartości innemu obiektowi mającemu uprawnienia do jej użycia — w tym przypadku innemu obiektowi typu `A` poprzez metodę `receiveD()`.

Interfejs `E` pokazuje, że interfejsy mogą być zagnieżdżane w sobie nawzajem. Jednakże reguły dotyczące interfejsów — w szczególności to, że ich elementy muszą być publiczne — są ściśle przestrzegane także w tym przypadku, a zatem interfejsy zagnieżdżone stają się automatycznie publiczne i nie mogą przez to być uczynione prywatnymi.

`NestingInterfaces` pokazuje różne sposoby implementowania interfejsów zagnieżdżonych. Warto zauważyć, że implementując interfejs, nie jesteśmy zmuszeni do implementowania interfejsów w nim zagnieżdżonych. Zwróćmy też uwagę, że interfejsy prywatne nie mogą być implementowane poza klasą, w której zostały zdefiniowane.

Początkowo może się wydawać, że wszystkie te elementy języka zostały dodane jedynie w celu składniowej jednorodności, jednak zwykle jest tak, że jeśli poznamy jakiś element języka, wcześniej czy później znajdziemy dla niego jakieś zastosowanie.

Interfejsy a wytwórnie

Interfejs ma być bramą prowadzącą do wielu implementacji; typowym sposobem tworzenia obiektów spełniających interfejs jest realizacja wzorca projektowego *Factory Method* („metoda wytwórcza"). Zamiast wprost wywoływać konstruktor, możemy wywoływać metodę obiektu-wytwórni, który generuje implementacje interfejsu — w ten sposób (przynajmniej teoretycznie) można całkowicie odizolować własny kod od implementacji interfejsu, co umożliwia transparentną wymianę jednej implementacji na inną. Oto program ilustrujący strukturę wzorca Factory Method:

```
//: interfaces/Factories.java
import static net.mindview.util.Print.*;

interface Service {
  void method1();
  void method2();
}
```

```
interface ServiceFactory {
  Service getService();
}

class Implementation1 implements Service {
  Implementation1() {} // Dostęp pakietowy
  public void method1() {print("Implementation1 method1");}
  public void method2() {print("Implementation1 method2");}
}

class Implementation1Factory implements ServiceFactory {
  public Service getService() {
    return new Implementation1();
  }
}

class Implementation2 implements Service {
  Implementation2() {} // Dostęp pakietowy
  public void method1() {print("Implementation2 method1");}
  public void method2() {print("Implementation2 method2");}
}

class Implementation2Factory implements ServiceFactory {
  public Service getService() {
    return new Implementation2();
  }
}

public class Factories {
  public static void serviceConsumer(ServiceFactory fact) {
    Service s = fact.getService();
    s.method1();
    s.method2();
  }
  public static void main(String[] args) {
    serviceConsumer(new Implementation1Factory());
    // Implementacje są całkowicie wymienne:
    serviceConsumer(new Implementation2Factory());
  }
} /* Output:
Implementation1 method1
Implementation1 method2
Implementation2 method1
Implementation2 method2
*///:~
```

Bez wzorca Factory Method gdzieś w kodzie trzeba by określić dokładny typ tworzonego obiektu Service, tak aby dało się wywołać odpowiedni konstruktor.

Po co nam jednak dodatkowy poziom pośredniości? Przydaje się on choćby przy tworzeniu szkieletów. Załóżmy, że tworzymy system rozgrywania gier pozwalający na przykład rozgrywać na tej samej planszy szachy i warcaby:

```
//: interfaces/Games.java
// Infrastruktura rozgrywek na planszy szachowej
// - wcielenie Factory Method.
import static net.mindview.util.Print.*;
```

```
interface Game { boolean move(); }
interface GameFactory { Game getGame(); }

class Checkers implements Game {
  private int moves = 0;
  private static final int MOVES = 3;
  public boolean move() {
    print("Warcaby: ruch " + moves);
    return ++moves != MOVES;
  }
}

class CheckersFactory implements GameFactory {
  public Game getGame() { return new Checkers(); }
}

class Chess implements Game {
  private int moves = 0;
  private static final int MOVES = 4;
  public boolean move() {
    print("Szachy: ruch " + moves);
    return ++moves != MOVES;
  }
}

class ChessFactory implements GameFactory {
  public Game getGame() { return new Chess(); }
}

public class Games {
  public static void playGame(GameFactory factory) {
    Game s = factory.getGame();
    while(s.move())
      ;
  }
  public static void main(String[] args) {
    playGame(new CheckersFactory());
    playGame(new ChessFactory());
  }
} /* Output:
Warcaby: ruch 0
Warcaby: ruch 1
Warcaby: ruch 2
Szachy: ruch 0
Szachy: ruch 1
Szachy: ruch 2
Szachy: ruch 3
*///:~
```

Gdyby klasa Games reprezentowała bardziej skomplikowany moduł kodu, to podejście pozwoliłoby na łatwe ponowne wykorzystanie tego kodu do kontrolowania rozgrywek różnych gier. Można sobie wyobrazić również bardziej skomplikowane gry czerpiące korzyści z takiego wzorca.

W następnym rozdziale zaprezentuję bardziej elegancki sposób implementowania wytwórni — za pośrednictwem anonimowych klas wewnętrznych.

Ćwiczenie 18. Utwórz interfejs `Cycle` z implementacjami w klasach `Unicycle`, `Bicycle` i `Tricycle`. Utwórz wytwórnie dla każdego konkretnego typu `Cycle` i kod, który używa tych wytwórni (2).

Ćwiczenie 19. Na bazie wzorca Factory Method utwórz infrastrukturę dla rozgrywek polegających na rzucaniu monetą i rzucaniu kośćmi (3).

Podsumowanie

W tym miejscu chciałoby się napisać, że interfejsy są świetne i powinno się je stosować zamiast konkretnych klas. Oczywiście niemal zawsze wybór pada na klasę, a zamiast tego można by utworzyć interfejs i wytwórnię.

Wielu programistów ulega tej pokusie, tworząc interfejsy i wytwórnie, gdzie tylko mogą. Zdaje się, że uzasadniają to oni możliwością wykorzystywania różnych implementacji, na którą to okoliczność zabezpieczają się abstrakcją w postaci interfejsu. Należy to jednak uznać za rodzaj przedwczesnej optymalizacji.

Istnienie wszelkich abstrakcji powinno wynikać z konkretnej potrzeby. Interfejsy to coś, co można wdrożyć do projektu, kiedy okaże się to niezbędne, niekoniecznie instalując ów dodatkowy poziom pośredniości — a wraz z nim dodatkowy poziom komplikacji — wszędzie, gdzie się da. Ów narzut złożoności jest wcale znaczący; zmuszanie programistów-klientów do borykania się z nim tylko dlatego, żeby przygotować się na „wszelki wypadek", raczej nie przyniesie pokłasku — gorzej, osobiście zaczynam wtedy poddawać w wątpliwość kompetencje twórcy w zakresie projektu.

Właściwą wytyczną jest *preferowanie klas wobec interfejsów*. Zaczynać należy od klas, a jeśli gdzieś ujawni się potrzeba wykorzystania interfejsu, można to zawsze zrobić. Interfejsy to świetne narzędzia, byle ich nie nadużywać.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 10.

Klasy wewnętrzne

Java pozwala na osadzanie definicji klasy w obrębie definicji innej klasy. Mowa wtedy o klasie wewnętrznej.

Klasy wewnętrzne są cenną cechą języka, umożliwiają bowiem grupowanie logicznie powiązanych ze sobą klas i kontrolowanie widoczności jednych w drugich. Istotne jest jednak zrozumienie, że są one czymś zupełnie innym niż kompozycje.

Na pierwszy rzut oka klasy wewnętrzne to jedynie prosty mechanizm ukrywania kodu przez osadzanie jednych klas w drugich. Przekonasz się jednak, że klasa wewnętrzna to coś więcej — ma bowiem wiedzę o klasie otaczającej i może się z nią komunikować. Przekonasz się też, że kod wykorzystujący klasy wewnętrzne może być bardziej elegancki i przejrzysty, choć samo ich zastosowanie nie jest oczywiście gwarancją czytelności kodu.

Przy pierwszym kontakcie klasy wewnętrzne sprawiają wrażenie dziwactwa, a żeby skutecznie wdrażać je w swoich projektach, trzeba się z nimi oswoić. Potrzeba istnienia klas wewnętrznych nie wydaje się często oczywista podczas uczenia się o nich, dlatego po zaprezentowaniu podstaw składni i semantyki klas wewnętrznych odpowiemy sobie na pytanie, jakie korzyści płyną z ich stosowania.

Reszta rozdziału poświęcona będzie tajnikom składni klas wewnętrznych. Przytaczane tam aspekty klas wewnętrznych niekoniecznie muszą być użyteczne w codziennej praktyce programistycznej. Być może przez jakiś czas nie będziesz tego potrzebował — lekturę tej części rozdziału możesz więc odłożyć na później, traktując ją jako podręcznik i dokumentację programistyczną.

Tworzenie klas wewnętrznych

Klasy wewnętrzne tworzy się tak, jak można by przypuszczać — poprzez umieszczenie ich definicji w otaczającej klasie:

```
//: innerclasses/Parcel1.java
// Tworzenie klas wewnętrznych.

public class Parcel1 {
  class Contents {
```

```
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) {
      label = whereTo;
    }
    String readLabel() { return label; }
  }
  // Klas wewnętrznych w klasie Parcel1
  // używamy tak jak wszelkich innych klas:
  public void ship(String dest) {
    Contents c = new Contents();
    Destination d = new Destination(dest);
    System.out.println(d.readLabel());
  }
  public static void main(String[] args) {
    Parcel1 p = new Parcel1();
    p.ship("Tasmania");
  }
} /* Output:
Tasmania
*///:~
```

W metodzie `ship()` wykorzystanie klas wewnętrznych wygląda tak jak wykorzystanie dowolnych innych klas. Praktycznie jedyną różnicą jest to, że nazwy są zagnieżdżone wewnątrz klasy `Parcel1`. Za moment zobaczymy, że nie jest to jedyna różnica.

W bardziej typowej sytuacji klasa zewnętrzna będzie miała metodę zwracającą referencję do klasy wewnętrznej, jak w tym przykładzie:

```
//: innerclasses/Parcel2.java
// Zwracanie referencji klasy wewnętrznej.

public class Parcel2 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) {
      label = whereTo;
    }
    String readLabel() { return label; }
  }
  public Destination to(String s) {
    return new Destination(s);
  }
  public Contents contents() {
    return new Contents();
  }
  public void ship(String dest) {
    Contents c = contents();
    Destination d = to(dest);
    System.out.println(d.readLabel());
  }
```

```
  public static void main(String[] args) {
    Parcel2 p = new Parcel2();
    p.ship("Tasmania");
    Parcel2 q = new Parcel2();
    // Definicje referencji klas wewnętrznych:
    Parcel2.Contents c = q.contents();
    Parcel2.Destination d = q.to("Borneo");
  }
} /* Output:
Tasmania
*///:~
```

Jeżeli chcemy stworzyć obiekt klasy wewnętrznej gdziekolwiek poza wnętrzem metody klasy zewnętrznej, typ obiektu musimy podać jako `NazwaKlasyZewnętrznej.NazwaKlasyZagnieżdżonej`, tak jak można było zaobserwować w metodzie `main()`.

Ćwiczenie 1. Napisz klasę o nazwie `Outer`, która będzie zawierać klasę wewnętrzną `Inner`. Dodaj do `Outer` metodę zwracającą obiekt typu `Inner`. W metodzie `main()` utwórz i zainicjalizuj referencję klasy `Inner` (1).

Połączenie z klasą zewnętrzną

Jak dotąd wydaje się, że klasy wewnętrzne stanowią jedynie sposób ukrywania nazw i organizacji kodu — są przydatne, ale nie powalają na kolana. Istnieje jednak jeszcze jeden „szczegół" ich dotyczący. Po stworzeniu obiekt klasy wewnętrznej zawiera specjalny *łącznik z obiektem klasy zewnętrznej*, który go stworzył, przez co ma dostęp do składowych tego otaczającego obiektu *bez żadnej specjalnej kwalifikacji*. Dodatkowo klasy wewnętrzne posiadają prawo dostępu do wszystkich elementów klasy je otaczającej[1]. Pokazuje to następujący przykład:

```
//: innerclasses/Sequence.java
// Klasa sekwencji obiektów Object.

interface Selector {
  boolean end();
  Object current();
  void next();
}

public class Sequence {
  private Object[] items;
  private int next = 0;
  public Sequence(int size) { items = new Object[size]; }
  public void add(Object x) {
    if(next < items.length)
      items[next++] = x;
  }
  private class SequenceSelector implements Selector {
    private int i = 0;
```

[1] To duża różnica w stosunku do *klas zagnieżdżonych* w C++, które są po prostu mechanizmem ukrywania nazw. Nie mają one łącznika do obiektu je zawierającego ani domyślnych praw dostępu do składowych.

```
    public boolean end() { return i == items.length; }
    public Object current() { return items[i]; }
    public void next() { if(i < items.length) i++; }
  }
  public Selector selector() {
    return new SequenceSelector();
  }
  public static void main(String[] args) {
    Sequence sequence = new Sequence(10);
    for(int i = 0; i < 10; i++)
      sequence.add(Integer.toString(i));
    Selector selector = sequence.selector();
    while(!selector.end()) {
      System.out.print(selector.current() + " ");
      selector.next();
    }
  }
} /* Output:
0 1 2 3 4 5 6 7 8 9
*///:~
```

Klasa Sequence jest po prostu „opakowaniem" tablicy referencji typu Object o ustalonym rozmiarze. W celu dodania elementu na koniec tej sekwencji wywołujemy add() (działa to dobrze tylko wtedy, gdy zostało jeszcze miejsce). Do pobrania któregoś z obiektów służy interfejs o nazwie Selector. To realizacja wzorca projektowego *Iterator*, o którym powiemy sobie więcej w dalszej części książki. Interfejs Selector pozwala na sprawdzenie, czy jesteśmy na końcu (metoda end()), przyjrzenie się aktualnemu obiektowi (metoda current()) oraz na przesunięcie się do następnego obiektu sekwencji (metoda next()). Ponieważ Selector jest interfejsem, może być implementowany przez wiele różnych klas, a wiele metod może pobierać go jako argument — wszystko w celu tworzenia bardziej ogólnego kodu.

W przykładzie SequenceSelector jest prywatną klasą posiadającą funkcje określone przez Selector. W metodzie main() możemy zaobserwować tworzenie obiektu Sequence, po którym następuje dodanie do reprezentowanej przez niego sekwencji kilku obiektów typu String. Następnie produkowany jest (z użyciem getSelector()) Selector używany następnie do przechodzenia przez wszystkie elementy sekwencji reprezentowanej przez obiekt typu Sequence.

Z początku tworzenie SequenceSelector wygląda tak, jak tworzenie zwykłej klasy wewnętrznej. Przyjrzyjmy się jednak dokładniej. Zauważmy, że każda z metod end(), current() i next() odwołuje się do referencji objects, nie będącej częścią SequenceSeletor, ale prywatnym polem klasy ją otaczającej. Klasa wewnętrzna może jednak odwoływać się do metod i pól klasy ją otaczającej, tak jak do własnych. Może się to okazać bardzo wygodne, jak można było zobaczyć w powyższym przykładzie.

A zatem klasa wewnętrzna ma automatycznie dostęp do składowych klasy ją zawierającej. Jak to możliwe? Egzemplarz klasy wewnętrznej musi przechowywać referencję do konkretnego obiektu klasy ją zawierającej — tego, który stworzył tę instancję. Następnie, podczas odnoszenia się do składnika klasy otaczającej, ta referencja jest wykorzystywana do wybrania go. Na szczęście kompilator bierze na siebie wszystkie szczegóły, musimy jednak pamiętać, że obiekt klasy wewnętrznej może być stworzony jedynie w powiązaniu z obiektem klasy ją zawierającej (o ile klasa wewnętrzna nie jest statyczna).

Konstrukcja obiektu klasy wewnętrznej wymaga referencji do obiektu klasy zewnętrznej, a kompilator zaprotestuje, jeżeli nie będzie mógł się do tej referencji dostać. Zwykle wszystko dzieje się jednak bez interwencji ze strony programisty.

Ćwiczenie 2. Napisz klasę przechowującą obiekt ciągu klasy `String` z metodą `toString()` wyświetlającą ten ciąg. Dodaj do obiektu Sequence kilka egzemplarzy swojej nowej klasy, a potem wypisz zawarte w nich ciągi na wyjściu programu (1).

Ćwiczenie 3. Zmodyfikuj ćwiczenie 1. tak, aby klasa `Outer` posiadała prywatne pole `String` (inicjalizowane w konstruktorze), a klasa `Inner` — metodę `toString()` wypisującą wartość tego pola na wyjściu. Utwórz obiekt klasy `Inner` i wywołaj jego metodę `toString()` (1).

.this i .new

Kiedy trzeba wygenerować referencję obicktu klasy zewnętrznej, obiekt ten określa się za pośrednictwem nazwy klasy zewnętrznej, kropki (.) i słowa kluczowego `this`. Tak powstała referencja automatycznie otrzymuje odpowiedni typ, znany i sprawdzany w czasie kompilacji, co eliminuje ewentualne narzuty czasu wykonania. Oto przykład użycia konstrukcji `.this`:

```
//: innerclasses/DotThis.java
// Kwalifikacja dostępu do obiektu klasy zewnętrznej.

public class DotThis {
  void f() { System.out.println("DotThis.f()"); }
  public class Inner {
    public DotThis outer() {
      return DotThis.this;
      // Zwykle "this" odnosiłoby się do obiektu klasy wewnętrznej
    }
  }
  public Inner inner() { return new Inner(); }
  public static void main(String[] args) {
    DotThis dt = new DotThis();
    DotThis.Inner dti = dt.inner();
    dti.outer().f();
  }
} /* Output:
DotThis.f()
*///:~
```

Niekiedy trzeba też poinstruować inny obiekt o konieczności utworzenia obiektu jednej z jego klas wewnętrznych. W tym celu trzeba w wyrażeniu operatora `new` podać referencję obiektu klasy zewnętrznej, za pośrednictwem zapisu `.new`, jak tu:

```
//: innerclasses/DotNew.java
// Tworzenie obiektu klasy wewnętrznej
// za pomocą zapisu .new syntax.

public class DotNew {
  public class Inner {}
```

```
  public static void main(String[] args) {
    DotNew dn = new DotNew();
    DotNew.Inner dni = dn.new Inner();
  }
} ///:~
```

Aby wprost utworzyć obiekt klasy wewnętrznej, należy odnieść się w kwalifikacji nie do nazwy klasy zewnętrznej, a do nazwy *obiektu* klasy zewnętrznej, jak powyżej. Powołanie się na egzemplarz klasy eliminuje również kwestię zasięgu nazw w klasach wewnętrznych, dzięki czemu nie trzeba pisać na przykład `dn.new DotNew.Inner()`.

Nie można utworzyć obiektu klasy wewnętrznej, jeśli nie istnieje jeszcze obiekt klasy zewnętrznej. Obiekt klasy wewnętrznej jest przecież niejawnie kojarzony z obiektem klasy zewnętrznej, w którego „łonie" powstaje. Nie dotyczy to kategorii *klas zagnieżdżonych* (czyli statycznych klas wewnętrznych) — tworzenie obiektów takich klas nie wymaga referencji obiektów klas zewnętrznych.

Oto jak zastosowalibyśmy zapis `.new` w przykładzie „Parcel":

```
// : innerclasses/Parcel3.java
// Użycie .new do tworzenia egzemplarz klas wewnętrznych.

public class Parcel3 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) { label = whereTo; }
    String readLabel() { return label; }
  }
  public static void main(String[] args) {
    Parcel3 p = new Parcel3();
    // Do utworzenia egzemplarza klasy wewnętrznej
    // konieczny jest egzemplarz klasy zewnętrznej:
    Parcel3.Contents c = p.new Contents();
    Parcel3.Destination d = p.new Destination("Tasmania");
  }
} ///:~
```

Ćwiczenie 4. Do klasy `Sequence.SequenceSelector` dodaj metodę wytwarzającą referencję obiektu klasy zewnętrznej (Sequence) (2).

Ćwiczenie 5. Utwórz klasę z klasą wewnętrzną. W osobnej klasie utwórz egzemplarz tamtej klasy wewnętrznej (1).

Klasy wewnętrzne a rzutowanie w górę

Klasy wewnętrzne naprawdę przydadzą się dopiero wtedy, gdy zaczniemy rzutować je do klasy bazowej, a w szczególności do interfejsu bazowego (efekt polegający na wyprodukowaniu referencji do interfejsu z obiektu implementującego go jest właściwie

tym samym co rzutowanie na klasę bazową). Dzieje się tak dlatego, że klasa zagnieżdżona — będąca implementacją interfejsu — może być całkowicie niewidoczna i niedostępna dla każdego, co jest wygodne przy ukrywaniu implementacji. Wszystko, co dostajemy, to referencja do interfejsu lub klasy bazowej.

Możemy niniejszym utworzyć interfejsy dla poprzednich przykładów:

```
//: innerclasses/Destination.java
public interface Destination {
  String readLabel();
} ///:~

//: innerclasses/Contents.java
public interface Contents {
  int value();
} ///:~
```

Interfejsy `Contents` i `Destination` są teraz dostępne dla programisty-klienta (pamiętajmy, że interfejs automatycznie czyni swoje składowe publicznymi).

Jeżeli otrzymamy referencje do bazowej klasy lub interfejsu, możliwe jest nawet, że nie będziemy w stanie odkryć dokładnego typu, tak jak tutaj:

```
//: innerclasses/TestParcel.java

class Parcel4 {
  private class PContents implements Contents {
    private int i = 11;
    public int value() { return i; }
  }
  protected class PDestination implements Destination {
    private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
  }
  public Destination destination(String s) {
    return new PDestination(s);
  }
  public Contents contents() {
    return new PContents();
  }
}

public class TestParcel {
  public static void main(String[] args) {
    Parcel4 p = new Parcel4();
    Contents c = p.contents();
    Destination d = p.destination("Tasmania");
    // Niedozwolone -- brak dostępu do klasy prywatnej:
    //! Parcel4.PContents pc = p.new PContents();
  }
} ///:~
```

W klasie Parcel4 zostało dodane coś nowego: klasa wewnętrzna PContents jest prywatna, a zatem nic poza Parcel4 nie ma do niej dostępu. Klasy zwykłe (niebędące wewnętrznymi) nie mogą być oznaczane jako prywatne czy zabezpieczone — dostęp do nich może być albo publiczny, albo pakietowy. PDestination jest chroniona, dlatego nie ma do niej dostępu nikt poza Parcel4, innymi klasami z tego samego co Parcel4 pakietu (ponieważ modyfikator protected daje również dostęp w obrębie pakietu) oraz klasami dziedziczącymi po Parcel4. Oznacza to, że programista-klient ma ograniczoną wiedzę i dostęp do tych składników. Nie można nawet rzutować w dół na prywatną lub chronioną (jeżeli jesteśmy w innym pakiecie i nie dziedziczymy po klasie zewnętrznej) klasę zagnieżdżoną, ponieważ nie mamy dostępu do jej nazwy, jak widać w przypadku klasy TestParcel. Prywatne klasy wewnętrzne umożliwiają zatem projektantowi klas całkowite uniemożliwienie kodowania uzależniającego od konkretnego typu, a przez to całkowite ukrycie implementacji. Rozszerzanie interfejsu jest w takim przypadku również bezużyteczne z punktu widzenia programisty-klienta, ponieważ nie ma on dostępu do dodatkowych metod nie będących częścią publicznego interfejsu. Dzięki stosowaniu tej techniki kompilator Javy ma także możliwość wyprodukowania bardziej efektywnego kodu.

Ćwiczenie 6. Stwórz interfejs z co najmniej jedną metodą umieszczony w swoim własnym pakiecie. W oddzielnym pakiecie stwórz klasę. Dodaj chronioną klasę wewnętrzną, implementującą interfejs. W trzecim pakiecie wywiedź przez dziedziczenie nową klasę oraz, wewnątrz jej metody, zwróć obiekt chronionej klasy wewnętrznej, podczas zwracania dokonując rzutowania w górę (2).

Ćwiczenie 7. Utwórz klasę z prywatnym polem i prywatną metodą. Umieść w niej klasę wewnętrzną z metodą modyfikującą pole klasy otaczającej i wywołującą metodę klasy otaczającej. W innej metodzie klasy otaczającej utwórz obiekt klasy wewnętrznej i wywołaj jego metodę, a potem pokaż wpływ tego wywołania na obiekt klasy otaczającej (2).

Ćwiczenie 8. Sprawdź, czy klasa zewnętrzna ma dostęp do prywatnych elementów klasy wewnętrznej (2).

Klasy wewnętrzne w metodach i zasięgach

To, co widzieliśmy do tej pory, obejmuje typowe wykorzystanie klas wewnętrznych. Zwykle kod, który będziemy pisać (i czytać), będzie obejmował jedynie „proste" klasy wewnętrzne — raczej łatwe do zrozumienia. Składnia klas wewnętrznych w Javie jest jednak bardzo rozbudowana i obejmuje wiele możliwości mniej oczywistych ich zastosowań: mogą one być tworzone wewnątrz metod, a nawet arbitralnie ustalonych zasięgów. Uzasadniają to dwie sytuacje:

1. Tak jak pokazano wcześniej, możemy implementować jakiś interfejs, aby umożliwić sobie stworzenie obiektu i zwrócenie referencji do niego.
2. Przy rozwiązywaniu złożonego problemu możemy potrzebować utworzenia pomocniczej klasy, której jednak nie chcielibyśmy czynić publicznie dostępną.

W następnych przykładach kod z poprzednich przykładów zostanie zmodyfikowany tak, aby zawierał:

1. Klasę zdefiniowaną wewnątrz metody.
2. Klasę zdefiniowaną wewnątrz zasięgu określonego w metodzie.
3. *Anonimową* klasę implementującą interfejs.
4. Anonimową klasę rozszerzającą klasę posiadającą konstruktor inny od domyślnego.
5. Anonimową klasę przeprowadzającą inicjalizację pól.
6. Anonimową klasę przeprowadzającą konstrukcję z wykorzystaniem inicjalizacji instancji (anonimowe klasy wewnętrzne nie mogą mieć konstruktorów).

Pierwszy przykład pokazuje tworzenie całej klasy w zasięgu wyznaczonym przez metodę (zamiast przez całą klasę). Klasa taka nosi nazwę *lokalnej klasy wewnętrznej*:

```
//: innerclasses/Parcel5.java
// Zagnieżdżanie klasy w metodzie.

public class Parcel5 {
  public Destination destination(String s) {
    class PDestination implements Destination {
      private String label;
      private PDestination(String whereTo) {
        label = whereTo;
      }
      public String readLabel() { return label; }
    }
    return new PDestination(s);
  }
  public static void main(String[] args) {
    Parcel5 p = new Parcel5();
    Destination d = p.destination("Tasmania");
  }
} ///:~
```

Klasa `PDestination` jest raczej częścią metody `dest()` niż klasy `Parcel5`. Dlatego też niemożliwy jest dostęp do `PDestination` spoza wnętrza `dest()`, z wyjątkiem dostępu do referencji typu `Destination`, typu bazowego. Oczywiście to, że definicja `PDestination` została umieszczona wewnątrz `dest()`, nie oznacza, że obiekty tego typu nie są ważne po powrocie z tej metody.

Identyfikatora `PDestination` moglibyśmy użyć dla klas zagnieżdżonych w kilku klasach z tego samego podkatalogu bez powodowania konfliktu nazw.

Następny przykład pokazuje, w jaki sposób można zagnieździć klasę w dowolnym, arbitralnie zdefiniowanym zasięgu:

```
//: innerclasses/Parcel6.java
// Zagnieżdżanie klasy w zasięgu.

public class Parcel6 {
  private void internalTracking(boolean b) {
    if(b) {
      class TrackingSlip {
```

```
        private String id;
        TrackingSlip(String s) {
          id = s;
        }
        String getSlip() { return id; }
      }
      TrackingSlip ts = new TrackingSlip("slip");
      String s = ts.getSlip();
    }
    // Ta klasa nie może być używana poza zasięgiem:
    //! TrackingSlip ts = new TrackingSlip("x");
  }
  public void track() { internalTracking(true); }
  public static void main(String[] args) {
    Parcel6 p = new Parcel6();
    p.track();
  }
} ///:~
```

Klasa `TrackingSlip` jest zagnieżdżona w zasięgu instrukcji warunkowej `if`. Nie znaczy to, że klasa jest tworzona warunkowo — kompiluje się ona tak samo, jak cała reszta programu. Nie jest jednak dostępna poza zasięgiem, w którym została zdefiniowana. Poza tym wygląda jak zwykła klasa.

Ćwiczenie 9. Stwórz interfejs z przynajmniej jedną metodą, a następnie zaimplementuj go poprzez zdefiniowanie klasy wewnętrznej w metodzie zwracającej Twój interfejs (1).

Ćwiczenie 10. Powtórz ćwiczenie 9., ale klasę wewnętrzną zdefiniuj w zasięgu zdefiniowanym wewnątrz metody (1).

Ćwiczenie 11. Stwórz prywatną klasę wewnętrzną, implementującą publiczny interfejs. Napisz metodę zwracającą referencję do instancji tej prywatnej klasy, a następnie rzutuj ją w górę do interfejsu. Wykaż, że klasa wewnętrzna jest całkowicie ukryta poprzez próbę rzutowania w dół (2).

Anonimowe klasy wewnętrzne

Kolejny przykład wygląda nieco dziwnie:

```
//: innerclasses/Parcel7.java
// Zwracanie egzemplarza anonimowej klasy wewnętrznej.

public class Parcel7 {
  public Contents contents() {
    return new Contents() { // Wstawienie definicji klasy
      private int i = 11;
      public int value() { return i; }
    }; // Tu wymagany średnik
  }
  public static void main(String[] args) {
    Parcel7 p = new Parcel7();
    Contents c = p.contents();
  }
} ///:~
```

Metoda `contents()` łączy tworzenie zwracanej wartości z definicją klasy reprezentowanej przez tę wartość! Klasa ta jest dodatkowo *anonimowa* — nie ma nazwy. Aby jeszcze nieco skomplikować sprawę, zaczyna się to tak, jak gdyby był tworzony obiekt typu `Contents`. Potem jednak, zanim dochodzimy do średnika, mówimy kompilatorowi: „Zaczekaj chwilę, wstawię tu chyba definicję klasy".

Ta dziwna składnia dosłownie oznacza: „Stwórz nowy obiekt anonimowej klasy dziedziczącej po `Contents`". Referencja zwracana przez wyrażenie `new` jest automatycznie rzutowana w górę do referencji typu `Contents`. Składnia anonimowych klas wewnętrznych stanowi skrót dla:

```
//: innerclasses/Parcel7b.java
// Rozwinięta wersja Parcel7.java

public class Parcel7b {
  class MyContents implements Contents {
    private int i = 11;
    public int value() { return i; }
  }
  public Contents contents() { return new MyContents(); }
  public static void main(String[] args) {
    Parcel7b p = new Parcel7b();
    Contents c = p.contents();
  }
} ///:~
```

W anonimowej klasie wewnętrznej obiekt `Contents` jest tworzony z użyciem konstruktora domyślnego.

Poniższy kod pokazuje, co należy robić, jeżeli konstruktor klasy bazowej wymaga argumentów:

```
//: innerclasses/Parcel8.java
// Wywołanie konstruktora klasy bazowej.

public class Parcel8 {
  public Wrapping wrapping(int x) {
    // Wywołanie konstruktora klasy bazowej:
    return new Wrapping(x) { // Przekazanie argumentu konstruktora.
      public int value() {
        return super.value() * 47;
      }
    }; // Wymagany średnik
  }
  public static void main(String[] args) {
    Parcel8 p = new Parcel8();
    Wrapping w = p.wrapping(10);
  }
} ///:~
```

Tak więc należy po prostu przekazać odpowiedni argument konstruktorowi klasy bazowej, tak jak jest przekazywany argument `x` w konstrukcji `new Wrapping(x)`. Choć `Wrapping` to najzwyklejsza klasa z implementacją, służy też jako wspólny „interfejs" dla klas pochodnych:

```
//: innerclasses/Wrapping.java
public class Wrapping {
  private int i;
  public Wrapping(int x) { i = x; }
  public int value() { return i; }
} ///:~
```

Zauważ, że `Wrapping` posiada konstruktor wymagający przekazania argumentu, co dodatkowo „uatrakcyjnia" składnię klasy anonimowej.

Średnik na końcu anonimowej klasy wewnętrznej nie oznaczał tu końca ciała klasy (jak to się dzieje w C++), lecz koniec wyrażenia przypadkiem zawierającego klasę anonimową. Jest to więc takie samo wykorzystanie średnika jak gdziekolwiek indziej.

Istnieje także możliwość inicjalizacji pól klas anonimowych w miejscu ich definicji:

```
//: innerclasses/Parcel9.java
// Anonimowa klasa wewnętrzna realizująca
// inicjalizację. Krótsza wersja Parcel5.java.

public class Parcel9 {
  // Argument używany w anonimowej klasie wewnętrznej
  // musi być finalny:
  public Destination destination(final String dest) {
    return new Destination() {
      private String label = dest;
      public String readLabel() { return label; }
    };
  }
  public static void main(String[] args) {
    Parcel9 p = new Parcel9();
    Destination d = p.destination("Tasmania");
  }
} ///:~
```

Gdy definiując anonimową klasę wewnętrzną, chcemy odwołać się do lokalnego obiektu metody zdefiniowanego poza tą klasą, musi to być obiekt finalny. Dlatego właśnie argument metody `destination()` jest poprzedzony przez `final`. Jeżeli o tym zapomnimy, w czasie kompilacji otrzymamy komunikat o błędzie.

Rozwiązanie to jest wystarczające, dopóki ograniczamy się do prostego przypisywania wartości polom. Co zrobić, gdy musimy przeprowadzić skomplikowaną inicjalizację w stylu konstruktora? W klasie anonimowej nie można utworzyć normalnego konstruktora, gdyż klasa nie ma żadnej nazwy; niemniej jednak, wykorzystując składnię *inicjalizacji egzemplarza*, możemy efektywnie stworzyć konstruktor dla anonimowej klasy wewnętrznej:

```
//: innerclasses/AnonymousConstructor.java
// Konstruktor dla anonimowej klasy wewnętrznej.
import static net.mindview.util.Print.*;

abstract class Base {
  public Base(int i) {
    print("Konstruktor bazowy, i = " + i);
  }
  public abstract void f();
}
```

```
public class AnonymousConstructor {
  public static Base getBase(int i) {
    return new Base(i) {
      { print("Wewnątrz inicjalizatora egzemplarza"); }
      public void f() {
        print("W metodzie f() klasy anonimowej");
      }
    };
  }
  public static void main(String[] args) {
    Base base = getBase(47);
    base.f();
  }
} /* Output:
Konstruktor bazowy, i = 47
Wewnątrz inicjalizatora egzemplarza
W metodzie f() klasy anonimowej
*///:~
```

W powyższym programie zmienna `i` *nie musi* być finalna. Choć jest ona przekazywana do konstruktora bazowego klasy anonimowej, to jednak *wewnątrz* tej klasy nigdy nie jest ona bezpośrednio używana.

Oto nowa wersja „paczki" wykorzystująca inicjalizację instancji. Należy zwrócić uwagę, że argumenty wywołania metody `destination()` muszą być finalne, gdyż są używane wewnątrz klasy anonimowej:

```
//: innerclasses/Parcel10.java
// Inicjalizacja egzemplarza w służbie konstrukcji
// anonimowej klasy wewnętrznej.

public class Parcel10 {
  public Destination
  destination(final String dest, final float price) {
    return new Destination() {
      private int cost;
      // Inicjalizacja egzemplarzowa dla każdego obiektu:
      {
        cost = Math.round(price);
        if(cost > 100)
          System.out.println("Za drogo!");
      }
      private String label = dest;
      public String readLabel() { return label; }
    };
  }
  public static void main(String[] args) {
    Parcel10 p = new Parcel10();
    Destination d = p.destination("Tasmania", 101.395F);
  }
} /* Output:
Za drogo!
*///:~
```

Wewnątrz inicjalizatora instancji widzimy kod, który nie mógłby zostać wykonany jako część inicjalizacji pola (tj. instrukcję `if`). W efekcie inicjalizator instancji stanowi konstruktor anonimowej klasy wewnętrznej. Rozwiązanie takie ma oczywiście swoje ograniczenia — nie możemy np. przeciążyć inicjalizatora instancji, możemy zatem mieć tylko jeden „konstruktor".

Anonimowe klasy wewnętrzne są cokolwiek ograniczone w porównaniu z klasycznym dziedziczeniem, ponieważ mogą albo rozszerzać klasę, albo implementować interfejs (nigdy razem). A w tym drugim przypadku implementacja dotyczy wyłącznie jednego interfejsu.

Ćwiczenie 12. Powtórz ćwiczenie 7. z użyciem anonimowej klasy wewnętrznej (1).

Ćwiczenie 13. Powtórz ćwiczenie 9. z użyciem anonimowej klasy wewnętrznej (1).

Ćwiczenie 14. Zmodyfikuj program *interfaces/HorrorShow.java*, implementując w nim klasy anonimowe `DangerousMonster` oraz `Vampire` (1).

Ćwiczenie 15. Stwórz klasę bez konstruktora domyślnego (bezargumentowego) i z jakimś innym konstruktorem (wymagającym podania argumentów). Stwórz drugą klasę z metodą zwracającą referencję do pierwszej klasy. Stwórz obiekt mający być zwrócony przez tę metodę poprzez utworzenie anonimowej klasy wewnętrznej, dziedziczącej po pierwszej z klas (2).

Jeszcze o wzorcu Factory Method

Spójrzmy, o ile przyjemniej prezentowałby się przykład *interfaces/Factories.java* z użyciem klas anonimowych:

```
// : innerclasses/Factories.java
import static net.mindview.util.Print.*;

interface Service {
  void method1();
  void method2();
}

interface ServiceFactory {
  Service getService();
}

class Implementation1 implements Service {
  private Implementation1() {}
  public void method1() {print("Implementation1 method1");}
  public void method2() {print("Implementation1 method2");}
  public static ServiceFactory factory =
    new ServiceFactory() {
      public Service getService() {
        return new Implementation1();
      }
    };
}
```

```
class Implementation2 implements Service {
  private Implementation2() {}
  public void method1() {print("Implementation2 method1");}
  public void method2() {print("Implementation2 method2");}
  public static ServiceFactory factory =
    new ServiceFactory() {
      public Service getService() {
        return new Implementation2();
      }
    };
}

public class Factories {
  public static void serviceConsumer(ServiceFactory fact) {
    Service s = fact.getService();
    s.method1();
    s.method2();
  }
  public static void main(String[] args) {
    serviceConsumer(Implementation1.factory);
    // Implementacje są całkowicie wymienne:
    serviceConsumer(Implementation2.factory);
  }
} /* Output:
Implementation1 method1
Implementation1 method2
Implementation2 method1
Implementation2 method2
*///:~
```

W tej wersji konstruktory klas `Implementation1` i `Implementation2` mogą być prywatne, nie ma też potrzeby tworzenia nazwanej klasy-wytwórni. Do tego często wytwórnia ma generować zaledwie jeden obiekt, co można zrealizować jak powyżej, a więc przez statyczne pole w implementacji interfejsu `Service`. Również otrzymana składnia jest wyrazistsza.

Anonimowe klasy wewnętrzne moglibyśmy też wykorzystać do udoskonalenia przykładu *interfaces/Games.java*:

```
//: innerclasses/Games.java
// Anonimowe klasy wewnętrzne w systemie rozgrywek planszowych.
import static net.mindview.util.Print.*;

interface Game { boolean move(); }
interface GameFactory { Game getGame(); }

class Checkers implements Game {
  private Checkers() {}
  private int moves = 0;
  private static final int MOVES = 3;
  public boolean move() {
    print("Warcaby: ruch " + moves);
    return ++moves != MOVES;
  }
  public static GameFactory factory = new GameFactory() {
    public Game getGame() { return new Checkers(); }
  };
}
```

```
class Chess implements Game {
  private Chess() {}
  private int moves = 0;
  private static final int MOVES = 4;
  public boolean move() {
    print("Szachy: ruch " + moves);
    return ++moves != MOVES;
  }
  public static GameFactory factory = new GameFactory() {
    public Game getGame() { return new Chess(); }
  };
}

public class Games {
  public static void playGame(GameFactory factory) {
    Game s = factory.getGame();
    while(s.move())
      ;
  }
  public static void main(String[] args) {
    playGame(Checkers.factory);
    playGame(Chess.factory);
  }
} /* Output:
Warcaby: ruch 0
Warcaby: ruch 1
Warcaby: ruch 2
Szachy: ruch 0
Szachy: ruch 1
Szachy: ruch 2
Szachy: ruch 3
*///:~
```

Pamiętajmy jednak o poradzie, która kończyła poprzedni rozdział, a dotyczącej preferowania klas wobec interfejsów. Jeśli dany projekt wymaga zastosowania interfejsu, będzie to wyraźnie widoczne. Jednak tam, gdzie interfejsy są zbędne, należy ich unikać.

Ćwiczenie 16. Zmień rozwiązanie ćwiczenia 18. z rozdziału „Interfejsy" tak, aby wykorzystywało anonimowe klasy wewnętrzne (1).

Ćwiczenie 17. Zmień rozwiązanie ćwiczenia 19. z rozdziału „Interfejsy" tak, aby wykorzystywało anonimowe klasy wewnętrzne (1).

Klasy zagnieżdżone

Jeżeli nie potrzebujemy połączenia pomiędzy obiektem klasy wewnętrznej a obiektem jego klasy zewnętrznej, możemy uczynić klasę wewnętrzną statyczną. Takie klasy są powszechnie określane jako *klasy zagnieżdżone*[2]. Aby zrozumieć znaczenie słowa kluczowego `static` w odniesieniu do klas wewnętrznych, musimy pamiętać, że obiekt zwykłej

[2] Przypominają one nieco klasy zagnieżdżone w języku C++, a jedyną różnicą jest to, że klasy zagnieżdżone w Javie mają dostęp do składowych prywatnych, a w C++ nie.

klasy wewnętrznej przechowuje niejawnie referencję do obiektu klasy otaczającej, który go stworzył. Nie jest to prawdą w przypadku, gdy klasa wewnętrzna zostanie poprzedzona słowem kluczowym `static`. Zagnieżdżenie klasy oznacza, że:

1. Nie potrzebujemy obiektu klasy zewnętrznej, aby stworzyć obiekt statycznej klasy wewnętrznej.
2. Z obiektu wewnętrznej klasy statycznej nie mamy dostępu do składników obiektu klasy zewnętrznej.

Jest jeszcze jedna różnica pomiędzy klasami zagnieżdżonymi a zwyczajnymi klasami wewnętrznymi. Pola i metody zwyczajnych klas wewnętrznych mogą być jedynie w zewnętrznym poziomie klasy, co znaczy, że takie klasy nie mogą mieć statycznych danych, pól i statycznych klas wewnętrznych. Klasy zagnieżdżone mogą natomiast mieć to wszystko:

```
//: innerclasses/Parcel11.java
// Klasy zagnieżdżone (statyczne klasy wewnętrzne).

public class Parcel11 {
  private static class ParcelContents implements Contents {
    private int i = 11;
    public int value() { return i; }
  }
  protected static class ParcelDestination
  implements Destination {
    private String label;
    private ParcelDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
    // Klasy zagnieżdżone mogą zawierać elementy statyczne:
    public static void f() {}
    static int x = 10;
    static class AnotherLevel {
      public static void f() {}
      static int x = 10;
    }
  }
  public static Destination destination(String s) {
    return new ParcelDestination(s);
  }
  public static Contents contents() {
    return new ParcelContents();
  }
  public static void main(String[] args) {
    Contents c = contents();
    Destination d = destination("Tasmania");
  }
} ///:~
```

W metodzie `main()` obiekt klasy `Parcel11` nie jest potrzebny, zamiast niego wykorzystujemy zwykłą składnię wyboru składnika statycznego do wywołania metod zwracających referencje typów `Contents` i `Destination`.

Jak się już przekonaliśmy, w zwykłej (niestatycznej) klasie wewnętrznej łącznik z klasą zewnętrzną jest realizowany przez specjalną referencję `this`. Klasa zagnieżdżona nie posiada takiej referencji, co pozwala porównać ją z metodą statyczną.

Ćwiczenie 18. Stwórz klasę zawierającą klasę zagnieżdżoną. W metodzie `main()` stwórz instancję klasy wewnętrznej (1).

Ćwiczenie 19. Stwórz klasę zawierającą klasę wewnętrzną, która z kolei sama zawiera inną klasę wewnętrzną. Powtórz to samo, używając klas zagnieżdżonych. Zwróć uwagę na nazwy plików *.class* wyprodukowanych przez kompilator (2).

Klasy wewnątrz interfejsów

Normalnie wewnątrz interfejsu nie wolno umieszczać żadnego kodu, można jednak w nim umieścić klasę zagnieżdżoną. Wszelki kod umieszczony wewnątrz interfejsu staje się automatycznie publiczny i statyczny. Ponieważ klasa jest statyczna, nie narusza zatem reguł dotyczących interfejsów — jest ona jedynie umieszczana w przestrzeni nazw interfejsu. W takiej klasie wewnętrznej można nawet zaimplementować zewnętrzny interfejs, jak tutaj:

```
// : innerclasses/ClassInInterface.java
// {main: ClassInInterface$Test}

public interface ClassInInterface {
  void howdy();
  class Test implements ClassInInterface {
    public void howdy() {
      System.out.println("Jak leci?");
    }
    public static void main(String[] args) {
      new Test().howdy();
    }
  }
} /* Output:
Jak leci?
*///:~
```

Wygodnie jest zagnieżdżać klasę wewnątrz interfejsu, kiedy chce się utworzyć wspólny kod, do użytku wszystkich implementacji tego interfejsu.

W jednym z poprzednich rozdziałów radziłem, aby w każdej klasie umieszczać metodę `main()`, która miałaby występować w roli stanowiska testowego klasy. Wadą takiego podejścia jest nadmiar dodatkowo kompilowanego kodu. Jeśli stanie się on problematyczny, kod testowy można ująć w klasie zagnieżdżonej:

```
// : innerclasses/TestBed.java
// Kod testowy w klasie zagnieżdżonej.
// {main: TestBed$Tester}

public class TestBed {
  public void f() { System.out.println("f()"); }
  public static class Tester {
    public static void main(String[] args) {
      TestBed t = new TestBed();
```

```
      t.f();
    }
  }
} /* Output:
f()
*///:~
```

W ten sposób powstanie osobna klasa o nazwie `TestBed$Tester` (aby uruchomić program, należy podać nazwę `TestBed$Tester`). Klasę można wykorzystać w celach testowych, nie trzeba jednak włączać jej do ostatecznej wersji kodu — wystarczy usunąć plik *TestBed$Tester.class* z przygotowywanego pakietu.

Ćwiczenie 20. Utwórz interfejs zawierający klasę zagnieżdżoną. Zaimplementuj interfejs i utwórz egzemplarz klasy zagnieżdżonej (1).

Ćwiczenie 21. Utwórz interfejs zawierający klasę zagnieżdżoną z metodą statyczną, która wywołuje metodę interfejsu i wypisuje wynik wywołania na wyjściu. Zaimplementuj interfejs i przekaż egzemplarz implementacji do metody (2).

Sięganie na zewnątrz z klasy wielokrotnie zagnieżdżonej

Bez względu na to, jak bardzo zagnieżdżona jest klasa, możemy w prosty sposób odwoływać się do wszystkich składowych klas, w których jest zagnieżdżona[3]. Widać to w przykładzie:

```
//: innerclasses/MultiNestingAccess.java
// Klasy zagnieżdżone mogą odwoływać się do wszystkich
// składowych wszystkich wyższych poziomów zagnieżdżenia.

class MNA {
  private void f() {}
  class A {
    private void g() {}
    public class B {
      void h() {
        g();
        f();
      }
    }
  }
}

public class MultiNestingAccess {
  public static void main(String[] args) {
    MNA mna = new MNA();
    MNA.A mnaa = mna.new A();
    MNA.A.B mnaab = mnaa.new B();
    mnaab.h();
  }
} ///:~
```

[3] Jeszcze raz dziękuję Martinowi Dannerowi.

Widzimy, że w MNA.A.B metody g() i f() są wywoływalne bez żadnej dodatkowej kwalifikacji (chociaż są one prywatne). Przykład ten demonstruje również składnię niezbędną do tworzenia obiektów klas wielokrotnie zagnieżdżonych z wnętrza innych klas. Składnia .new dostarcza właściwy zasięg, dlatego nie musimy stosować kwalifikacji nazwy klasy w wywołaniu konstruktora.

Dlaczego klasy wewnętrzne

Jak dotąd poznaliśmy wiele składni i semantyki opisującej sposób, w jaki działają, nie odpowiedzieliśmy jednak na pytanie o cel ich istnienia. Dlaczego twórcy Javy zdecydowali się na duży przecież wysiłek dodania tak fundamentalnego elementu języka?

Zazwyczaj klasa wewnętrzna dziedziczy z innej klasy lub implementuje interfejs oraz manipuluje danymi klasy, w której jest zawarta. Można powiedzieć, że klasa wewnętrzna jest rodzajem okna na klasę ją zawierającą.

Pytanie dotyczące istoty klas wewnętrznych brzmi: „Skoro potrzebuję jedynie referencji do pewnego interfejsu, to dlaczego nie implementuje go po prostu klasa zewnętrzna?". Jeżeli rzeczywiście jest to wszystko, czego potrzebujemy, powinniśmy to zrobić w ten sposób. Jaka jest więc różnica między klasą wewnętrzną, implementującą interfejs, a klasą zewnętrzną również go implementującą? Otóż nie zawsze mamy komfort pracy z interfejsami — czasami musimy działać na implementacjach. Najważniejszym uzasadnieniem wprowadzenia klas wewnętrznych jest więc to, że:

> *każda klasa wewnętrzna może niezależnie dziedziczyć implementację. A zatem klasy takie nie są ograniczone przez fakt, że klasa zewnętrzna dziedziczy już jakąś implementację.*

Bez dostarczanej przez klasy wewnętrzne efektywnej możliwości dziedziczenia po więcej niż jednej klasie konkretnej lub abstrakcyjnej wiele problemów projektowych i programistycznych byłoby bardzo trudnych. Zatem jednym ze sposobów patrzenia na klasy wewnętrzne jest traktowanie ich jako uzupełnienia rozwiązania problemu wielokrotnego dziedziczenia. Interfejsy rozwiązują część problemu, ale klasy wewnętrzne efektywnie umożliwiają „wielokrotne dziedziczenie implementacji", tzn. efektywnie umożliwiają dziedziczenie po więcej niż jednej rzeczy nie będącej interfejsem.

Aby lepiej to zrozumieć, rozważmy sytuację, gdy dwa interfejsy mają być w jakiś sposób zaimplementowane wewnątrz klasy. Dzięki elastyczności interfejsów mamy dwie możliwości — pojedyncza klasa lub klasa wewnętrzna:

```
// : innerclasses/MultiInterfaces.java
// Dwie metody implementowania w klasie wielu interfejsów.
package innerclasses;

interface A {}
interface B {}

class X implements A, B {}

class Y implements A {
```

```
  B makeB() {
    // Anonimowa klasa wewnętrzna:
    return new B() {};
  }
}

public class MultiInterfaces {
  static void takesA(A a) {}
  static void takesB(B b) {}
  public static void main(String[] args) {
    X x = new X();
    Y y = new Y();
    takesA(x);
    takesA(y);
    takesB(x);
    takesB(y.makeB());
  }
} ///:~
```

Oczywiście zakładamy, że oba rozwiązania wpisują się logicznie w strukturę naszego kodu. Zwykle jednak natura rozwiązywanego problemu daje nam jakieś wskazówki dotyczące wyboru pomiędzy pojedynczą klasą a klasą wewnętrzną. Jeśli jednak nie bierzemy pod uwagę zewnętrznych ograniczeń, to, które z rozwiązań z powyższego przykładu wybierzemy, nie ma z implementacyjnego punktu widzenia większego znaczenia. Oba będą działać.

Jeżeli jednak zamiast interfejsów mamy abstrakcyjne lub konkretne klasy, które nasza klasa musi w jakiś sposób implementować, jesteśmy nagle ograniczeni do zastosowania klas wewnętrznych:

```
//: innerclasses/MultiImplementation.java
// W klasach konkretnych i abstrakcyjnych klasy wewnętrzne
// są jedynym sposobem uzyskania efektu "wielodziedziczenia
// implementacji".
package innerclasses;

class D {}
abstract class E {}

class Z extends D {
  E makeE() { return new E() {}; }
}

public class MultiImplementation {
  static void takesD(D d) {}
  static void takesE(E e) {}
  public static void main(String[] args) {
    Z z = new Z();
    takesD(z);
    takesE(z.makeE());
  }
} ///:~
```

Jeżeli nie musimy rozwiązywać problemu „wielokrotnego dziedziczenia implementacji", możemy wyobrazić sobie programowanie bez użycia klas wewnętrznych. Dają nam one jednak dodatkowe możliwości:

1. Klasy wewnętrzne mogą mieć wiele instancji, każdą z odmienną informacją o stanie, niezależną od informacji przechowywanej w obiekcie klasy zewnętrznej.
2. W pojedynczej klasie zewnętrznej możemy mieć kilka klas wewnętrznych implementujących ten sam interfejs lub dziedziczących po tej samej klasie na różne sposoby. Przykład tego poznamy wkrótce.
3. Miejsce stworzenia obiektu klasy wewnętrznej nie jest związane z miejscem stworzenia klasy zewnętrznej.
4. Nie istnieje potencjalnie myląca relacja „bycia czymś" w stosunku do klasy wewnętrznej — jest ona bytem oddzielnym.

Gdybyśmy nie stosowali klas wewnętrznych w przykładzie *Sequence.java*, musielibyśmy zapisać w kodzie, że „sekwencja jest selektorem (`Sequence implements Selector`)", a dla każdego obiektu klasy `Sequence` moglibyśmy powołać do istnienia tylko jeden `Selector`. W zastosowanym przez nas rozwiązaniu możemy dla przykładu dodać metodę `getRSelector()`, tworzącą `Selector` poruszający się po sekwencji od końca. Tego rodzaju elastyczność jest możliwa jedynie z użyciem klas wewnętrznych.

Ćwiczenie 22. Zaimplementuj metodę `reverseSelector()` w pliku *Sequence.java* (2).

Ćwiczenie 23. Stwórz interfejs `U` posiadający trzy metody. Stwórz klasę `A` z metodą produkującą referencję do `U` poprzez zbudowanie anonimowej klasy wewnętrznej, następnie stwórz drugą klasę `B` zawierającą tablicę referencji `U`. `B` powinno mieć metodę akceptującą i przechowującą w tablicy referencję do `U`, drugą metodę ustawiającą referencję (podaną jako argument) na `null` oraz trzecią metodę, przechodzącą przez tablicę i wywołującą metody `U`. W `main()` stwórz grupę obiektów typu `A` i jeden obiekt typu `B`. Wypełnij `B` referencjami typu `U` wyprodukowanymi przez obiekty `A`. Wykorzystaj `B` do zwrotnego wywołania obiektów `A`. Usuń niektóre z referencji typu `U` z `B` (4).

Domknięcia i wywołania zwrotne

Domknięcie (ang. *closure*) jest wywoływalnym obiektem, przechowującym informację z miejsca, w którym został stworzony. Z definicji tej możemy wywnioskować, że klasy wewnętrzne są obiektowo zorientowanymi domknięciami. Nie przechowują one po prostu każdego fragmentu informacji z klasy zewnętrznej („miejsca, w którym zostały stworzone"), lecz automatycznie utrzymują referencję do całego obiektu klasy zewnętrznej, posiadając jednocześnie uprawnienia do manipulowania wszystkimi jego składnikami, łącznie z prywatnymi.

Jednym z najmocniejszych argumentów przemawiających za włączeniem do Javy jakiegoś rodzaju wskaźników było umożliwienie *wywołań zwrotnych* (ang. *callbacks*). Wywołanie zwrotne umożliwia przekazanie obiektowi pewnej informacji pozwalającej mu na późniejsze odwołanie się do obiektu, który informacje tę przekazał. Jak zobaczymy w dalszej części książki, stwarza to ogromne możliwości. Jeżeli jednak wywołanie zwrotne jest zaimplementowane z użyciem wskaźników, przed niewłaściwym ich wykorzystaniem chroni nas jedynie dobre wychowanie programisty. Jak już zdążyliśmy zauważyć, Java stara się być bardziej rozważna, dlatego wskaźniki nie zostały włączone do języka.

Domknięcie dostarczane przez klasę wewnętrzną stanowi tu idealne rozwiązanie — bardziej elastyczne i znacznie bezpieczniejsze od wskaźnika. Oto przykład:

```
//: innerclasses/Callbacks.java
// Klasy wewnętrzne w wywołaniach zwrotnych
package innerclasses;
import static net.mindview.util.Print.*;

interface Incrementable {
  void increment();
}

// Uproszczona klasa implementacji interfejsu:
class Callee1 implements Incrementable {
  private int i = 0;
  public void increment() {
    i++;
    print(i);
  }
}

class MyIncrement {
  public void increment() { print("Inna operacja"); }
  static void f(MyIncrement mi) { mi.increment(); }
}

// Jeśli dana klasa musi jakoś inaczej implementować
// metodę increment(), należy użyć klasy wewnętrznej:
class Callee2 extends MyIncrement {
  private int i = 0;
  public void increment() {
    super.increment();
    i++;
    print(i);
  }
  private class Closure implements Incrementable {
    public void increment() {
      // Określenie metody zewnętrznej zapobiegające
      // nieskończonej rekurencji wywołań:
      Callee2.this.increment();
    }
  }
  Incrementable getCallbackReference() {
    return new Closure();
  }
}

class Caller {
  private Incrementable callbackReference;
  Caller(Incrementable cbh) { callbackReference = cbh; }
  void go() { callbackReference.increment(); }
}

public class Callbacks {
  public static void main(String[] args) {
    Callee1 c1 = new Callee1();
    Callee2 c2 = new Callee2();
    MyIncrement.f(c2);
```

```
    Caller caller1 = new Caller(c1);
    Caller caller2 = new Caller(c2.getCallbackReference());
    caller1.go();
    caller1.go();
    caller2.go();
    caller2.go();
  }
} /* Output:
Inna operacja
1
1
2
Inna operacja
2
Inna operacja
3
*///:~
```

Widać tu też kolejną różnicę pomiędzy implementowaniem interfejsu w klasie zewnętrznej a robieniem tego samego w klasie wewnętrznej. W kategoriach kodu rozwiązanie reprezentowane przez `Callee1` jest zdecydowanie prostsze. `Callee2` dziedziczy po klasie `MyIncrement`, mającej już własną, inną metodę `increment()`, robiącą coś zupełnie innego niż to, czego spodziewamy się po interfejsie `Incrementable`. Dziedzicząc po `MyIncrement`, klasa `Callee2` nie może przesłonić używanej przez `Incrementable` metody `increment()`, a zatem jesteśmy zmuszeni do dostarczenia oddzielnej implementacji z użyciem klasy wewnętrznej. Zauważmy również, że tworząc klasę wewnętrzną, niczego nie dodajemy ani nie modyfikujemy w interfejsie klasy zewnętrznej.

W klasie `Callee2` prywatne jest wszystko z wyjątkiem metody `getCallbackReference()`. Interfejs `Incrementable` ma kluczowe znaczenie dla umożliwienia *jakiegokolwiek* dostępu do świata zewnętrznego. Oto w jaki sposób interfejsy Javy umożliwiają całkowite oddzielenie interfejsu od implementacji.

Klasa wewnętrzna `Closure` implementuje po prostu `Incrementable`, aby dostarczyć punkt zaczepienia do klasy `Callee2` — jest to jednak bezpieczny punkt zaczepienia. Każdy, kto dostanie referencję typu `Incrementable`, będzie oczywiście mógł wywołać metodę `increment()` — i nic poza tym (inaczej niż w przypadku wskaźnika, który pozwala na wiele innych rzeczy).

Klasa `Caller` pobiera referencję typu `Incrementable` w swym konstruktorze (choć przyjęcie referencji umożliwiającej wywołania zwrotne mogłoby się zdarzyć w dowolnym momencie), a potem, po jakimś czasie, wykorzystuje ją, aby „wywołać" klasę `Callee`.

Wartość wywołań zwrotnych polega na ich elastyczności — możemy dynamicznie określić, jakie funkcje zostaną wywołane w czasie wykonania. Zaleta tej możliwości będzie bardziej wyraźna w rozdziale „Graficzne interfejsy użytkownika", w którym wywołania zwrotne będą używane wszędzie dla zaimplementowania funkcji graficznego interfejsu użytkownika (ang. *graphical user interface*, GUI).

Klasy wewnętrzne a szkielety sterowania

Bardziej konkretny przykład wykorzystania klas wewnętrznych znajdziemy w czymś, co będziemy nazywać *szkieletem sterowania* (ang. *control framework*).

Szkielet aplikacji (ang. *application framework*) to klasa lub zestaw klas zaprojektowanych w celu rozwiązywania określonych problemów. Aby zastosować taki szkielet, dziedziczymy po jednej lub wielu klasach i przesłaniamy niektóre metody. Kod umieszczony w przesłoniętych metodach dostosowuje ogólne rozwiązanie zapewniane przez szkielet aplikacji do naszego specyficznego problemu (jest to przykład wzorca projektowego *Template Method*, informacje na jego temat można znaleźć w książce *Thinking in Patterns (with Java)* do pobrania z *www.MindView.net*). Wzorzec projektowy Template Method (*metoda szablonowa*) określa podstawową strukturę algorytmu i zakłada wywoływanie jednej albo wielu dających się przesłaniać metod, które doprecyzowują algorytm. Wzorzec projektowy oddziela to, co zmienne, od tego, co stałe; w tym przypadku stała jest właśnie ta struktura algorytmu, a elementami zmiennymi są przesłaniane metody wywoływane z wnętrza metody szablonowej.

Szkielet sterowania jest szczególnym przykładem szkieletu aplikacji, zdominowanym przez konieczność obsługi zdarzeń. System, którego główna działalność polega na reagowaniu na zdarzenia, nazywamy *systemem sterowanym zdarzeniami* (ang. *event-driven system*). Typowym problemem związanych z tworzeniem aplikacji jest zaś graficzny interfejs użytkownika (GUI), który jest niemal całkowicie sterowany zdarzeniami. W rozdziale „Graficzne interfejsy użytkownika" poznamy przykład szkieletu sterowania przewidzianego właśnie do roli rozwiązania problemu implementacji GUI, jakim jest biblioteka Swing. Wykorzystuje ona klasy wewnętrzne w bardzo intensywny sposób.

Aby zrozumieć, w jaki sposób klasy wewnętrzne pozwalają na proste tworzenie szkieletów sterowania, rozważymy przykład takiego szkieletu, którego zadaniem będzie wykonywanie zdarzeń w chwili, gdy staną się one „gotowe". Choć „gotowość" mogłaby oznaczać cokolwiek, w tym jednak przypadku jej podstawą będzie czas zegarowy. Poniżej widzimy szkielet sterowania niezawierający żadnej informacji na temat tego, czym steruje. Informacje te są określane w procesie dziedziczenia, w momencie implementacji zmiennej części algorytmu (`action()`) wywoływanej ze struktury algorytmu.

Po pierwsze, poniżej przedstawiony został interfejs opisujący dowolne zdarzenie sterowania. Nie jest on właściwym interfejsem, lecz klasą abstrakcyjną, ponieważ domyślnym zachowaniem jest sterowanie na podstawie czasu, możliwe jest zatem zamieszczenie pewnej domyślnej implementacji:

```
//: innerclasses/controller/Event.java
// Metody wspólne dla wszystkich zdarzeń sterowania.
package innerclasses.controller;

public abstract class Event {
  private long eventTime;
  protected final long delayTime;
  public Event(long delayTime) {
    this.delayTime = delayTime;
    start();
  }
```

```
  public void start() { // Umożliwia restart
    eventTime = System.nanoTime() + delayTime;
  }
  public boolean ready() {
    return System.nanoTime() >= eventTime;
  }
  public abstract void action();
} ///:~
```

Konstruktor pobiera tu po prostu czas (względem czasu utworzenia obiektu), w jakim chcielibyśmy uruchomić zdarzenie, a następnie wywołuje metodę `start()`, która pobiera aktualny czas i dodaje do niego podany odstęp, uzyskując w ten sposób czas, w jakim zdarzenie ma zajść. Możliwości funkcjonalne metody `start()` nie zostały umieszczone w konstruktorze obiektu, lecz zaimplementowano je jako niezależną metodę. Dzięki temu można bowiem powtórnie wykorzystać obiekt `Event`, gdy wcześniejsze zdarzenie zostało już wykonane. Na przykład, aby stworzyć zdarzenie zachodzące cyklicznie, wystarczy wywołać metodę `start()` wewnątrz metody `action()`.

Metoda `ready()` informuje, czy nadszedł już czas wywołania metody `action()`. Metoda `ready()` może być oczywiście przesłonięta w klasie potomnej w celu oparcia uruchamiania zdarzenia na podstawie innych rzeczy.

Poniższy plik zawiera właściwy szkielet sterowania, zarządzający zdarzeniami i uruchamiający je. Obiekty `Event` są przechowywane wewnątrz obiektu kontenera klasy `List<Event>` („listy obiektów `Event`"), który zostanie dokładniej przedstawiony w rozdziale „Kolekcje obiektów". Na razie wystarczy wiedzieć, że metoda `add()` dodaje obiekty (klasy `Event`) na końcu kontenera, `size()` — zwraca liczbę elementów przechowywanych w kontenerze, `get()` — zwraca element o podanym indeksie, a `remove()` usuwa z kontenera wybrany element.

```
// innerclasses/controller/Controller.java
// Szkielet systemów sterowania.
package innerclasses.controller;
import java.util.*;

public class Controller {
  // Klasa z biblioteki java.util do przechowywania obiektów Event:
  private List<Event> eventList = new ArrayList<Event>();
  public void addEvent(Event c) { eventList.add(c); }
  public void run() {
    while(eventList.size() > 0)
      // Utworzenie kopii listy, aby przy wybieraniu
      // elementów listy nie modyfikować oryginału:
      for(Event e : new ArrayList<Event>(eventList))
        if(e.ready()) {
          System.out.println(e);
          e.action();
          eventList.remove(e);
        }
  }
} ///:~
```

Metoda `run()` cyklicznie przegląda zawartość kopii listy `eventList` w poszukiwaniu zdarzeń gotowych do „zgłoszenia". W przypadku odnalezienia takiego zdarzenia informacje na jego temat są wyświetlane przy użyciu metody `toString()`, następnie wywoływana jest metoda `action()`, a sam obiekt jest usuwany z listy.

Zauważmy, że w naszym projekcie nie wiemy nic o tym, *co* właściwie robi dane zdarzenie. Na tym właśnie polega istota projektu — to, w jaki sposób „oddziela on rzeczy, które się zmieniają, od rzeczy, które pozostają takie same". Na „wektor zmian" — by użyć mojego określenia — składają się akcje różnego typu obiektów Event. Innymi słowy, różne akcje wyrażamy poprzez stworzenie różnych podklas klasy Event.

Tu wchodzą do gry klasy wewnętrzne. Umożliwiają one dwie rzeczy:

1. Stworzenie całej implementacji aplikacji na podstawie szkieletu sterowania w postaci pojedynczej klasy, dzięki czemu możliwa jest hermetyzacja wszystkiego, co jest specyficzne dla implementacji. Klasy wewnętrzne są tu wykorzystywane do wyrażenia wielu różnych rodzajów akcji (`action()`) niezbędnych do rozwiązania problemu. W podanym nieco dalej przykładzie używa się dodatkowo wewnętrznych klas prywatnych, dzięki czemu implementacja jest całkowicie ukryta i może być bezkarnie zmieniana.
2. Uniknięcie skomplikowanego i nieporęcznego kodu dzięki możliwości łatwego dostępu do składników klasy zewnętrznej. Bez tego kod mógłby stać się nieprzyjemny do tego stopnia, że zaczęlibyśmy rozglądać się za jakimś alternatywnym rozwiązaniem.

Rozważmy szczególną implementację szkieletu sterowania, zaprojektowaną w celu kontroli funkcjonowania szklarni[4]. Każda akcja jest tu zupełnie inna: włączanie i wyłączanie świateł, wody i termostatów, uruchamianie dzwoneczków i restartowanie systemu. Szkielet sterowania został jednak zaprojektowany do prostego izolowania tego różnorodnego kodu. Klasy wewnętrzne umożliwiają posiadanie wielu pochodnych wersji tej samej klasy bazowej Event wewnątrz jednej klasy. Dla każdego typu akcji wyprowadzamy więc nową wewnętrzną podklasę klasy Event, a następnie piszemy kod sterujący wewnątrz metody `action()`.

Tak jak to się zwykle robi ze szkieletami aplikacji, klasa `GreenhouseControls` (kontrola szklarni) dziedziczy po klasie `Controller`:

```
// : innerclasses/GreenhouseControls.java
// Konkretne wdrożenie systemu sterowania w postaci pojedynczej
// klasy. Klasy wewnętrzne pozwalają na hermetyzowanie różnych
// funkcjonalności dla poszczególnych typów zdarzeń.
import innerclasses.controller.*;

public class GreenhouseControls extends Controller {
  private boolean light = false;
  public class LightOn extends Event {
    public LightOn(long delayTime) { super(delayTime); }
    public void action() {
      // Tutaj kod sterowania sprzętem, który
      // fizycznie włączy światła.
      light = true;
    }
    public String toString() { return "Światła włączone"; }
  }
  public class LightOff extends Event {
```

[4] Z jakichś powodów rozwiązywanie tego problemu zawsze sprawiało mi przyjemność. Pochodzi on z mojej wcześniejszej książki *C++ Inside & Out*, jednak Java umożliwia znacznie prostsze rozwiązanie.

```
    public LightOff(long delayTime) { super(delayTime); }
    public void action() {
      // Tutaj kod sterowania sprzętem, który
      // fizycznie wyłączy światła.
      light = false;
    }
    public String toString() { return "Światła wyłączone"; }
  }
  private boolean water = false;
  public class WaterOn extends Event {
    public WaterOn(long delayTime) { super(delayTime); }
    public void action() {
      // Tu kod sterowania sprzętem.
      water = true;
    }
    public String toString() {
      return "Obieg wody w szklarni włączony";
    }
  }
  public class WaterOff extends Event {
    public WaterOff(long delayTime) { super(delayTime); }
    public void action() {
      // Tu kod sterowania sprzętem.
      water = false;
    }
    public String toString() {
      return "Obieg wody w szklarni wyłączony";
    }
  }
  private String thermostat = "Dzień";
  public class ThermostatNight extends Event {
    public ThermostatNight(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Tu kod sterowania sprzętem.
      thermostat = "Noc";
    }
    public String toString() {
      return "Termostat przełączony na ustawienie nocne";
    }
  }
  public class ThermostatDay extends Event {
    public ThermostatDay(long delayTime) {
      super(delayTime);
    }
    public void action() {
      // Tu kod sterowania sprzętem.
      thermostat = "Dzień";
    }
    public String toString() {
      return "Termostat przełączony na ustawienie dzienne";
    }
  }
  // Przykład metody action() uzupełniającej
  // listę zdarzeń o nowy obiekt zdarzenia
  public class Bell extends Event {
    public Bell(long delayTime) { super(delayTime); }
    public void action() {
```

```
      addEvent(new Bell(delayTime));
    }
    public String toString() { return "Dzyń!"; }
  }
  public class Restart extends Event {
    private Event[] eventList;
    public Restart(long delayTime, Event[] eventList) {
      super(delayTime);
      this.eventList = eventList;
      for(Event e : eventList)
        addEvent(e);
    }
    public void action() {
      for(Event e : eventList) {
        e.start(); // Ponowne wyzwolenie każdego zdarzenia
        addEvent(e);
      }
      start(); // Ponowne wyzwolenie danego zdarzenia
      addEvent(this);
    }
    public String toString() {
      return "Restart systemu";
    }
  }
  public static class Terminate extends Event {
    public Terminate(long delayTime) { super(delayTime); }
    public void action() { System.exit(0); }
    public String toString() { return "Koniec"; }
  }
} ///:~
```

Zauważmy, że mimo iż `light`, `water` i `thermostat` należą do zewnętrznej klasy `GreenhouseControls`, to jednak klasy wewnętrzne mają do tych pól dostęp bez dodatkowej kwalifikacji i specjalnych uprawnień. Poza tym większość metod `action()` zawiera jakąś kontrolę sprzętu.

Większość klas `Event` wygląda podobnie, jednak `Bell` (dzwonek) i `Restart` są wyjątkowe. Dzwonek (`Bell`) dzwoni, po czym dodaje kolejny obiekt `Bell` do listy zdarzeń, aby zadzwonić później kolejny raz. Zauważmy, jak dobrze klasy zagnieżdżone *udają* wielokrotne dziedziczenie: `Bell` oraz `Restart` posiadają wszystkie metody klasy `Event` i wydają się również posiadać wszystkie metody zewnętrznej klasy `GreenhouseControls`.

Obiekt `Restart` pobiera tablicę obiektów `Event`, które są następnie dodawane do systemu sterującego. Ponieważ obiekt `Restart` sam jest pewnym typem zdarzenia (`Event`), można go podać w wywołaniu `Restart.action()`, dzięki czemu system będzie się sam regularnie restartować.

Poniższa klasa konfiguruje system, tworząc obiekt `GreenhouseControls` i dodając do niego różne rodzaje obiektów `Event`. Jest to przykład wzorca projektowego o nazwie *Command* (*polecenie*) — każdy obiekt z listy `eventList` jest żądaniem, poleceniem hermetyzowanym w obiekcie:

```
//: innerclasses/GreenhouseController.java
// Konfiguracja i uruchomienie systemu osługi szklarni.
// {Args: 5000}
import innerclasses.controller.*;
```

```
public class GreenhouseController {
  public static void main(String[] args) {
    GreenhouseControls gc = new GreenhouseControls();
    // Zamiast sztywnego konfigurowania komponentów możemy
    // tu wczytać parametry startowe z pliku tekstowego:
    gc.addEvent(gc.new Bell(900));
    Event[] eventList = {
      gc.new ThermostatNight(0),
      gc.new LightOn(200),
      gc.new LightOff(400),
      gc.new WaterOn(600),
      gc.new WaterOff(800),
      gc.new ThermostatDay(1400)
    };
    gc.addEvent(gc.new Restart(2000, eventList));
    if(args.length == 1)
      gc.addEvent(
        new GreenhouseControls.Terminate(
          new Integer(args[0])));
    gc.run();
  }
} /* Output:
Dzyń!
Termostat przełączony na ustawienie nocne
Światła włączone
Światła wyłączone
Obieg wody w szklarni włączony
Obieg wody w szklarni wyłączony
Termostat przełączony na ustawienie dzienne
Restart systemu
Koniec
*///:~
```

Powyższa klasa inicjalizuje system, tak że wszystkie niezbędne zdarzenia są do niego dodawane. Następuje cykliczne wyzwalanie zdarzenia restartu systemu (`Restart`), a w jego ramach wczytanie listy zdarzeń (`eventList`) do obiektu `GreenhouseControls`. Jeśli w wierszu wywołania programu podamy argument wyrażony w milisekundach, po upływie określonego czasu program zostanie przerwany (to element infrastruktury testowej).

Oczywiście bardziej elastycznym sposobem wykonania powyższych czynności jest odczytanie informacji o zdarzeniach z pliku, a nie podawanie ich na stałe w kodzie programu (jedno z ćwiczeń z rozdziału „Wejście-wyjście" polega właśnie na wprowadzeniu takich modyfikacji).

Powyższy przykład powinien ułatwić docenienie wartości klas wewnętrznych, szczególnie w odniesieniu do szkieletów sterowania. Jednakże w rozdziale „Graficzne interfejsy użytkownika" zobaczymy, jak elegancko można wykorzystać klasy wewnętrzne do opisania akcji w graficznym interfejsie użytkownika. Po lekturze tamtego rozdziału będziesz już całkowicie przekonany do sensu istnienia klas wewnętrznych.

Ćwiczenie 24. W przykładzie *GreenhouseControls.java* dodaj wewnętrzne podklasy klasy `Event` odpowiedzialne za włącznie i wyłączanie wentylatorów. Skonfiguruj program *GreenhouseController.java* tak, aby korzystał z tych nowych obiektów (2).

Ćwiczenie 25. Stwórz klasę dziedziczącą po `GreenhouseControls` (zdefiniowanej w *GreenhouseControls.java*), aby dodać wewnętrzną klasę `Event` włączającą i wyłączającą generatory mgły. Napisz nową wersję programu *GreenhouseController.java*, która będzie wykorzystywać te nowe obiekty (3).

Dziedziczenie po klasach wewnętrznych

Ponieważ konstruktor klasy wewnętrznej musi podłączyć referencję do obiektu otaczającego, zatem przy dziedziczeniu po klasie wewnętrznej sprawy nieco się komplikują. Problem polega na tym, że „sekretna" referencja do obiektu obejmującego *musi* zostać zainicjalizowana, a w przypadku klasy pochodnej nie ma już domyślnego obiektu, do którego należy się podłączyć. Rozwiązaniem jest użycie poniższej składni do jawnego określenia powiązania:

```
//: innerclasses/InheritInner.java
// Dziedziczenie po klasie wewnętrznej.

class WithInner {
  class Inner {}
}

public class InheritInner extends WithInner.Inner {
  //! InheritInner() {} // Nie da się skompilować
  InheritInner(WithInner wi) {
    wi.super();
  }
  public static void main(String[] args) {
    WithInner wi = new WithInner();
    InheritInner ii = new InheritInner(wi);
  }
} ///:~
```

Można zauważyć, że `InheritInner` rozszerza jedynie klasę wewnętrzną, nie zaś zewnętrzną. Jeżeli jednak przychodzi do stworzenia konstruktora, okazuje się, że domyślny nie jest dobry i że nie możemy po prostu przekazać referencji do obiektu obejmującego. Musimy dodatkowo użyć składni:

```
referencjaDoKlasyObejmującej.super();
```

wewnątrz konstruktora. Zapewnia to niezbędną referencję i program się już skompiluje.

Ćwiczenie 26. Stwórz klasę z klasą wewnętrzną posiadającą inny od domyślnego konstruktor (wymagający podania argumentów). Stwórz drugą klasę zawierającą drugą klasę wewnętrzną, dziedziczącą po pierwszej klasie wewnętrznej (2).

Czy klasy wewnętrzne mogą być przesłaniane?

Co dzieje się, gdy tworzymy klasę wewnętrzną, dziedziczymy po klasie ją zawierającej, a następnie przedefiniujemy klasę wewnętrzną? Tzn. czy możliwe jest przesłonięcie klas wewnętrznych? Może to wyglądać na rozwiązanie o dużych możliwościach, ale tak naprawdę przesłanianie klasy wewnętrznej, jak gdyby była metodą, nie pełni żadnej specjalnej funkcji:

```
// : innerclasses/BigEgg.java
// Klasa wewnętrzna nie może być przesłaniana jak metoda.
import static net.mindview.util.Print.*;

class Egg {
  private Yolk y;
  protected class Yolk {
    public Yolk() { print("Egg.Yolk()"); }
  }
  public Egg() {
    print("Nowy obiekt Egg()");
    y = new Yolk();
  }
}

public class BigEgg extends Egg {
  public class Yolk {
    public Yolk() { print("BigEgg.Yolk()"); }
  }
  public static void main(String[] args) {
    new BigEgg();
```

ny automatycznie przez kompilator, wywołuje domyślny
omyśleć, że skoro tworzymy obiekt klasy BigEgg,
rsja klasy Yolk, jednak tak się nie dzieje.

wej „magii" związanej z klasami we-
mującej. Dwie klasy wnętrzne są
znajduje się w swojej przestrzeni nazw.
wewnętrznej po innej klasie wewnętrznej:

```
olk()"); }
2.Yolk.f()");}
```

```
  }
  private Yolk y = new Yolk();
  public Egg2() { print("Nowy obiekt Egg2()"); }
  public void insertYolk(Yolk yy) { y = yy; }
  public void g() { y.f(); }
}

public class BigEgg2 extends Egg2 {
  public class Yolk extends Egg2.Yolk {
    public Yolk() { print("BigEgg2.Yolk()"); }
    public void f() { print("BigEgg2.Yolk.f()"); }
  }
  public BigEgg2() { insertYolk(new Yolk()); }
  public static void main(String[] args) {
    Egg2 e2 = new BigEgg2();
    e2.g();
  }
} /* Output:
Egg2.Yolk()
Nowy obiekt Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
*///:~
```

W tym przykładzie `BigEgg2.Yolk` jawnie rozszerza `Egg2.Yolk` i przesłania jej metody. Metoda `insertYolk()` klasy `Egg2` pozwala klasie `BigEgg2` rzutować jeden z obiektów jej własnego podtypu `Yolk` do oryginalnego `Yolk` w celu umieszczenia referencji do niego w `y`. A zatem przy wywołaniu przez `g()` metody `y.f()` zostaje wywołana przesłonięta wersja tej ostatniej. Drugie wywołanie `Egg2.Yolk` miało miejsce jako wywołanie konstruktora klasy bazowej w konstruktorze `BigEgg2.Yolk`. Widzimy, że podczas działania `g()` została wywołana przesłonięta wersja `f()`.

Lokalne klasy wewnętrzne

Jak już wspominałem, klasy wewnętrzne można tworzyć także wewnątrz bloków kodu — w typowym przypadku wewnątrz metod. Lokalne klasy wewnętrzne nie mogą mieć żadnych specyfikatorów dostępu, gdyż nie stanowią one części klasy zewnętrznej, dysponują jednak dostępem do zmiennych finalnych w bloku, w którym zostały zdefiniowane, oraz do wszystkich składowych klasy zewnętrznej. Poniższy przykład porównuje wykorzystanie lokalnej klasy wewnętrznej oraz anonimowej klasy wewnętrznej:

```
//: innerclasses/LocalInnerClass.java
// Przechowuje sekwencję obiektów klasy Object.
import static net.mindview.util.Print.*;

interface Counter {
  int next();
}

public class LocalInnerClass {
  private int count = 0;
  Counter getCounter(final String name) {
```

```
    // Lokalna klasa wewnętrzna:
    class LocalCounter implements Counter {
      public LocalCounter() {
        // Lokalna klasa wewnętrzna może mieć konstruktor
        print("LocalCounter()");
      }
      public int next() {
        printnb(name); // Dostęp do lokalnego pola finalnego
        return count++;
      }
    }
    return new LocalCounter();
  }
  // Analogicznie w anonimowych klasach wewnętrznych:
  Counter getCounter2(final String name) {
    return new Counter() {
      // Anonimowa klasa wewnętrzna nie może posiadać nazwanego
      // konstruktora, a jedynie blok inicjalizacji egzemplarza
      {
        print("Counter()");
      }
      public int next() {
        printnb(name); // Dostęp do lokalnego pola finalnego
        return count++;
      }
    };
  }
  public static void main(String[] args) {
    LocalInnerClass lic = new LocalInnerClass();
    Counter
      c1 = lic.getCounter("Lokalny obiekt wewnętrzny "),
      c2 = lic.getCounter2("Anonimowy obiekt wewnętrzny ");
    for(int i = 0; i < 5; i++)
      print(c1.next());
    for(int i = 0; i < 5; i++)
      print(c2.next());
  }
} /* Output:
LocalCounter()
Counter()
Lokalny obiekt wewnętrzny 0
Lokalny obiekt wewnętrzny 1
Lokalny obiekt wewnętrzny 2
Lokalny obiekt wewnętrzny 3
Lokalny obiekt wewnętrzny 4
Anonimowy obiekt wewnętrzny 5
Anonimowy obiekt wewnętrzny 6
Anonimowy obiekt wewnętrzny 7
Anonimowy obiekt wewnętrzny 8
Anonimowy obiekt wewnętrzny 9
*///:~
```

zwraca kolejną wartość sekwencji. W przykładzie został on zaimple-
wno jako klasa lokalna, jak również jako anonimowa klasa wewnętrzna.
sponują takimi samymi zachowaniami i możliwościami. Ponieważ nazwa
wewnętrznej nie jest dostępna poza metodą, jedynym czynnikiem prze-
ıa korzyść tego rozwiązania (a nie użycia anonimowej klasy wewnętrznej)

jest konieczność posiadania normalnego, nazwanego konstruktora oraz przeciążania konstruktorów. Rozwiązania te nie są bowiem dostępne w anonimowych klas wewnętrznych, w których można stosować jedynie inicjalizację instancji.

Jedynym powodem przemawiającym za użyciem lokalnej klasy wewnętrznej (a nie anonimowej klasy wewnętrznej) jest konieczność wykorzystania więcej niż jednego obiektu danej klasy.

Identyfikatory klas wewnętrznych

Ponieważ dla każdej klasy powstaje plik *.class*, przechowujący informację o tym, jak stworzyć obiekt danego typu (na podstawie tej informacji tworzona jest „metaklasa" nazywana obiektem typu `Class`), możemy zatem przypuszczać, że także klasy wewnętrzne muszą generować odpowiednie pliki *.class*, zawierające informacje o *ich* obiektach `Class`. Nazwy tych plików określane są przez następującą regułę: najpierw dajemy nazwę klasy obejmującej, następnie znak `$`, a na koniec nazwę klasy wewnętrznej. Na przykład pliki *.class* stworzone przez przykład `InheritInner` obejmują:

```
Counter.class
LocalInnerClass$2.class
LocalInnerClass$1LocalCounter.class
LocalInnerClass.class
```

Jeżeli klasy są anonimowe, kompilator zaczyna po prostu używać kolejnych cyfr w charakterze ich identyfikatorów. Jeżeli klasy wewnętrzne są zagnieżdżone wewnątrz innych klas wewnętrznych, wtedy ich nazwy są dołączane po znaku `$` do identyfikatora klasy je obejmującej.

Mimo prostoty i naturalności takiego schematu tworzenia wewnętrznych nazw jest on solidny i obejmuje większość sytuacji[5]. Ponieważ schemat ten jest także standardem Javy, wygenerowane pliki są automatycznie niezależne od platformy (weźmy pod uwagę, że kompilator zmienia nasze klasy wewnętrzne także na wiele innych sposobów w celu umożliwienia ich działania).

Podsumowanie

Interfejsy i klasy wewnętrzne stanowią koncepcje bardziej wyrafinowane od tego, co znajdziemy w większości języków programowania zorientowanych obiektowo. W C++ na przykład nie znajdujemy niczego do nich podobnego. Wspólnie rozwiązują one ten sam problem, jaki C++ stara się rozwiązać z użyciem swego wielokrotnego dziedziczenia. Jednak wielokrotne dziedziczenie C++ okazuje się być trudniejsze w użyciu, podczas gdy interfejsy i klasy wewnętrzne Javy są znacznie przystępniejsze.

[5] Z drugiej strony, `$` należy do metaznaków powłoki systemu Unix, dlatego czasami będziemy mieli kłopoty z listowaniem plików *.class*. Jest to nieco dziwne posunięcie ze strony firmy Sun bazującej na Uniksie. Być może chodzi o to, iż ten temat nie był w ogóle rozważany przez projektantów, którzy pomyśleli, że będziemy skupiać się jedynie na plikach z kodem źródłowym.

Chociaż omawiane tu elementy języka są same w sobie dosyć naturalne, to jednak ich używanie jest kwestią projektowania — bardzo podobnie jak polimorfizm. Z czasem zaczniemy lepiej rozpoznawać sytuacje, w których należy użyć interfejsu, klasy wewnętrznej lub ich obu. Na tym etapie lektury powinniśmy przynajmniej zrozumieć składnię i semantykę. Omawiane elementy języka przyswoimy sobie ostatecznie, widząc je w działaniu.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 11.

Kolekcje obiektów

Program obejmujący wyłącznie ustaloną liczbę obiektów, których czas życia jest znany, to program dosyć prosty.

Generalnie nasze programy zawsze będą tworzyć nowe obiekty, wykorzystując pewne reguły, które będą znane dopiero w czasie wykonania. Przed uruchomieniem nie będziemy wiedzieć, ile potrzebujemy obiektów, ani nawet znać ich dokładnego typu. Aby zaradzić temu powszechnemu problemowi programistycznemu, potrzebna jest możliwość tworzenia dowolnej liczby obiektów, w dowolniej chwili i miejscu. Tak więc nie można polegać na tworzeniu referencji o określonej nazwie do utrzymywania każdego z obiektów:

```
MójTyp mojaReferencja;
```

ponieważ nigdy nie wiemy, ile ich rzeczywiście będzie potrzebnych.

W celu rozwiązania tego istotnego problemu Java pozwala przechowywać obiekty (albo raczej referencje do obiektów) na kilka sposobów. Wbudowanym typem jest tablica, o której pisałem wcześniej. Tablica sprawdza się najlepiej przy przechowywaniu grup obiektów, a także i wartości typów podstawowych. Jednak jej rozmiar jest stały, a przecież w ogólnym przypadku w czasie pisania programu nie wiemy jeszcze, ilu obiektów będziemy potrzebować, ani czy nie zajdzie potrzeba skorzystania z bardziej wyszukanych sposobów przechowywania obiektów — z tego punktu widzenia zwykła tablica jest zbyt ograniczona.

Biblioteka użytkowa Javy (`java.util`) zawiera dopracowany zbiór *klas kontenerowych*, z których najważniejsze to `List`, `Set`, `Queue` i `Map`. Klasy te znane są również jako *kolekcje*, lecz ponieważ biblioteki Java stosują określenie `Collection` w celu oznaczenia konkretnego podzbioru tej biblioteki, będę używał bardziej ogólnego terminu „kontener". Kontenery oferują wyszukane środki przechowywania obiektów, stanowiąc niemal gotowe rozwiązania zaskakująco wielu problemów programistycznych.

Obok wielu innych cech — przykładowo zbiór typu `Set` może zawierać tylko jeden obiekt o danej wartości, a `Map`, będący *tablicą asocjacyjną*, pozwala powiązać dowolny obiekt z dowolnym innym obiektem — wszystkie klasy kontenerowe Javy przejawiają wygodną cechę automatycznego dopasowania swojego rozmiaru do potrzeb. Tak więc w przeciwieństwie do tablic można włożyć dowolną liczbę obiektów, nie martwiąc się tym, jak duży pojemnik zadeklarować podczas pisania programu.

Choć klasy kontenerowe nie doczekały się obsługi wprost w języku Java[1], za pośrednictwem wyróżnionego słowa kluczowego, stanowią podstawowe narzędzia wspomagające programistę. W tym rozdziale przedstawię podstawowe informacje o kontenerach Javy, z naciskiem na typowe ich zastosowania. Skupimy się na tych kontenerach, które są najczęściej wykorzystywane w codziennej praktyce programistycznej. Później, w rozdziale „Kontenery z bliska", zaprezentuję resztę informacji o kontenerach i podam dodatkowe dane o technikach ich stosowania.

Kontenery typowane i uogólnione

Jednym z problemów trapiących programistów korzystających z implementacji kontenerów w wersjach poprzedzających wydanie Javy SE5 była możliwość wstawiania do kontenerów obiektów niewłaściwych typów. Weźmy za przykład kontener obiektów `Apple`; niech będzie to najbardziej podstawowy z kontenerów, czyli `ArrayList`. Póki co możesz go sobie wyobrażać jako „tablicę, która automatycznie zwiększa swój rozmiar". Zastosowanie `ArrayList` jest proste: najpierw należy ją stworzyć, włożyć obiekty, stosując metodę `add()`, a potem wydobyć je dzięki `get()`, podając indeks — tak jak w tablicy, ale bez nawiasów kwadratowych[2]. `ArrayList` posiada również metodę `size()`, która informuje o liczbie wstawionych do niej elementów, dzięki czemu można się zabezpieczyć przed nieumyślnym odwołaniem do nieistniejącego indeksu, a tym samym spowodowaniem błędu (objawiającego się *wyjątkiem czasu wykonania*; wyjątki omówimy w rozdziale „Obsługa błędów za pomocą wyjątków").

Wracając do przykładu, w kontenerze będziemy umieszczać obiekty klas `Apple` i `Orange`, a potem je z niego wydobywać. Normalnie kompilator Javy wystosowałby ostrzeżenie spowodowane tym, że na razie w przykładzie *nie* korzystamy z typów ogólnych (ang. *generics*). W SE5 ostrzeżenie to można zignorować specjalną adnotacją. *Adnotacje* (ang. *annotation*) zaczynają się od znaku `@` i mogą przyjmować argument; tu mowa o adnotacji `@SuppressWarnings`, z argumentem „unchecked", który wymusza ignorowanie ostrzeżeń kompilatora typu „unchecked":

```
//: holding/AppleAndOrangesWithoutGenerics.java
// Prosty przykład z kontenerem (prowokuje ostrzeżenia kompilatora).
// {ThrowsException}
import java.util.*;

class Apple {
  private static long counter;
  private final long id = counter++;
  public long id() { return id; }
}

class Orange {}
```

[1] W wielu językach programowania kontenery elementami wbudowanymi — dotyczy to choćby Perla, Pythona czy Ruby.

[2] Tu właśnie przydałaby się możliwość przeciążenia operatora. W C++ i C# klasy kontenerowe udostępniają elegancką składnię przeciążania operatorów.

```
public class ApplesAndOrangesWithoutGenerics {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    ArrayList apples = new ArrayList();
    for(int i = 0; i < 3; i++)
      apples.add(new Apple());
    // Niezabezpieczony przed pomieszaniem jabłek z pomarańczami:
    apples.add(new Orange());
    for(int i = 0; i < apples.size(); i++)
      ((Apple)apples.get(i)).id();
      // Obecność niepożądanego obiektu Orange jest
      // wykrywana dopiero w czasie wykonania
  }
} /* (Uruchom, by zobaczyć wyjście) *///:~
```

O adnotacjach dostępnych w SE5 dowiesz się więcej z rozdziału „Adnotacje".

Klasy `Apple` i `Orange` są zupełnie różne; poza tym, że obie dziedziczą (pośrednio) po klasie `Object`, nie mają ze sobą nic wspólnego (pamiętaj, że jeśli nie podasz jawnie klasy, po której dziedziczy dana klasa, to automatycznie stanie się ona pochodną klasy `Object`). A ponieważ kontener `ArrayList` przechowuje obiekty `Object`, to za pomocą metody `add()` kontenera można do niego dodawać nie tylko pożądane obiekty `Apple`, ale i obiekty `Orange`, i to bez próby zablokowania takiego wstawiania czy to w czasie kompilacji, czy w czasie wykonania. Kiedy potem za pomocą metody `get()` wydobywamy obiekty z kontenera, sądząc, że to egzemplarze `Apple`, otrzymamy referencje klasy `Object`, które musimy jawnie rzutować na typ `Apple`. Całe wyrażenie rzutowania trzeba ująć w nawiasy, aby na rzecz wynikowego obiektu (domniemanej klasy `Apple`) wywołać metodę `id()` — brak nawiasowania będzie błędem składniowym.

Tymczasem w czasie działania programu przy próbie rzutowania obiektu `Orange` na typ `Apple` dojdzie do błędu, objawiającego się wspomnianym już wyjątkiem.

W rozdziale „Typy ogólne" zasmakujesz skomplikowanych technik *tworzenia* klas za pośrednictwem typów ogólnych. Ale samo stosowanie gotowej klasy uogólnionej jest zwykle proste. Na przykład, aby zdefiniować kontener `ArrayList` przeznaczony do przechowywania obiektów `Apple`, należy zadeklarować go jako obiekt typu `ArrayList<Apple>` (zamiast zwykłego `Apple`). Nawias kątowy zawiera tak zwane *argumenty typowe* (może być ich wiele), dookreślające typ elementów przechowywanych przez dany egzemplarz kontenera.

Dzięki typom ogólnym można już *w czasie kompilacji* blokować próby umieszczania w kontenerze obiektów niewłaściwego typu[3]. Oto poprzedni przykład zmieniony pod kątem typów ogólnych:

```
//: holding/ApplesAndOrangesWithGenerics.java
import java.util.*;

public class ApplesAndOrangesWithGenerics {
  public static void main(String[] args) {
    ArrayList<Apple> apples = new ArrayList<Apple>();
```

[3] Pod koniec rozdziału „Typy ogólne" znajduje się dyskusja nad tym, czy to aby taki wielki problem. Tak czy inaczej, tamten rozdział pokaże również, że typy ogólne są w Javie przydatne nie tylko w obrębie kontenerów.

```
    for(int i = 0; i < 3; i++)
      apples.add(new Apple());
    // Błąd kompilacji:
    // apples.add(new Orange());
    for(int i = 0; i < apples.size(); i++)
      System.out.println(apples.get(i).id());
    // Składnia foreach:
    for(Apple c : apples)
      System.out.println(c.id());
  }
} /* Output:
0
1
2
0
1
2
*///:~
```

Tym razem kompilator uniemożliwi wstawienie do kontenera `apples` obiektu `Orange` — próba zostanie wychwycona w czasie kompilacji, a nie w czasie wykonania.

Zauważ też, że przy wydobywaniu obiektów z kontenera nie trzeba już rzutować ich na docelowy typ. Kontener zna przecież dokładny typ przechowywanych obiektów i może zwracać z `get()` referencję odpowiedniego typu niewymagającą rzutowania. Jak widać, typy ogólne pozwalają nie tylko na kontrolę typów w czasie kompilacji, ale i na wygodniejsze odwoływanie się do obiektów przechowywanych w kontenerze.

Omawiany przykład pokazuje też, że tam gdzie nie trzeba posługiwać się indeksami elementów kontenera, można przeglądać zawartość kontenera `ArrayList` za pomocą składni foreach.

Określenie typu obiektów przechowywanych za pomocą argumentu typowego nie oznacza ścisłego ograniczenia typu wstawianych elementów; rzutowanie w górę działa dla typów ogólnych tak samo jak dla wszelkich innych typów:

```
//: holding/GenericsAndUpcasting.java
import java.util.*;

class GrannySmith extends Apple {}
class Gala extends Apple {}
class Fuji extends Apple {}
class Braeburn extends Apple {}

public class GenericsAndUpcasting {
  public static void main(String[] args) {
    ArrayList<Apple> apples = new ArrayList<Apple>();
    apples.add(new GrannySmith());
    apples.add(new Gala());
    apples.add(new Fuji());
    apples.add(new Braeburn());
    for(Apple c : apples)
      System.out.println(c);
  }
} /* Output: (Sample)
GrannySmith@7d772e
```

```
Gala@11b86e7
Fuji@35ce36
Braeburn@757aef
*///:~
```

Jak widać, do kontenera deklarowanego jako przechowujący obiekty `Apple` można z powodzeniem wstawiać obiekty typów wyprowadzanych z `Apple`.

Wyjście programu jest generowane za pośrednictwem domyślnej wersji metody `toString()` zdefiniowanej dla klasy `Object`; metoda ta wypisuje nazwę klasy uzupełnioną szesnastkową reprezentacją *skrótu* (ang. *hash code*) identyfikującego dany egzemplarz klas (skrót generowany jest wywołaniem metody `hashCode()`). O owych skrótach powiemy sobie więcej w rozdziale „Kontenery z bliska".

Ćwiczenie 1. Utwórz nową klasę o nazwie `Gerbil` (myszoskoczek) ze składową `int gerbilNumber`, inicjalizowaną z poziomu konstruktora. Wyposaż klasę w metodę `hop()`, wypisującą wartość tej składowej i sygnalizującej wykonywanie podskoków. Utwórz kontener `ArrayList` i wstaw do niego obiekty `Gerbil`. Teraz skorzystaj z metody `get()` kontenera w celu przejrzenia jego zawartości i wywołania metody `hop()` dla każdego umieszczonego w nim myszoskoczka (2).

Pojęcia podstawowe

Biblioteka kontenerów Java 2 podejmuje wyzwanie „przechowywania naszych obiektów" i dzieli je na dwie dziedziny pojęciowe, wyrażane dwoma podstawowymi interfejsami biblioteki:

1. Kolekcja `Collection`: grupa odrębnych elementów, podlegająca jakimś regułom. Na przykład lista typu `List` musi przechowywać elementy w określonej kolejności, zbiór `Set` nie może zawierać duplikatów elementów, a kolejka `Queue` *porządkuje* elementy wedle *dyscypliny kolejki* (zwykle postuluje ona kolejność identyczną z kolejnością wstawiania elementów).
2. Odwzorowanie `Map`: grupa par obiektów typu klucz-wartość, pozwalająca na wydobywanie wartości dla znanego klucza. Wspominany już kontener `ArrayList` kojarzy co prawda z wszystkimi przechowywanymi elementami numer (indeks), więc realizuje odwzorowanie numerów do obiektów. Kontener `Map` pozwala jednak na odwzorowywanie obiektów do *innych obiektów*. Kontener ten zwany jest często *tablicą asocjacyjną* (bo kojarzy obiekty z innymi obiektami) albo *słownikiem* (bo udostępnia obiekty wartości dla podanych obiektów kluczy, jak przy wyszukiwaniu definicji dla danego hasła). Odwzorowania stanowią potężne narzędzie programistyczne.

Większość kodu będzie odwoływała się do kontenerów za pośrednictwem tych interfejsów, a jedyne miejsce wskazania dokładnego typu kontenera to miejsce jego utworzenia. Kontener `List` (listę) można stworzyć tak:

```
List<Apple> apples = new ArrayList<Apple>();
```

Zwróć uwagę na rzutowanie `ArrayList` w górę, na typ `List`, czego nie było w poprzednich przykładach. Celem stosowania ogólniejszego interfejsu jest zostawienie furtki dla ewentualnej przyszłej zmiany implementacji — wystarczy wtedy zmienić definicję w miejscu utworzenia kontenera, jak tutaj:

```
List<Apple> apples = new LinkedList<Apple>();
```

Całość polega na utworzeniu obiektu konkretnej klasy kontenera, rzutowaniu go w górę hierarchii na odpowiedni interfejs i korzystanie z tego interfejsu we wszystkich odwołaniach do kontenera.

To podejście nie zawsze działa, a to dlatego, że niektóre klasy kontenerowe cechują się rozszerzoną funkcjonalnością. Na przykład klasa `LinkedList` udostępnia metody nieobecne w interfejsie `List`, a klasa `TreeMap` ma metody spoza interfejsu `Map`. Kiedy chcemy z nich skorzystać, musimy zrezygnować z rzutowania w górę na bardziej uniwersalny interfejs.

Interfejs `Collection` („kolekcja") uogólnia pojęcie *sekwencji* jako sposobu grupowania obiektów. Oto prosty przykład wypełniania kolekcji (reprezentowanej tu kontenerem `ArrayList`) obiektami typu `Integer`, a następnie wypisania wartości wszystkich elementów wynikowego kontenera:

```
//: holding/SimpleCollection.java
import java.util.*;

public class SimpleCollection {
  public static void main(String[] args) {
    Collection<Integer> c = new ArrayList<Integer>();
    for(int i = 0; i < 10; i++)
      c.add(i); // Automatyczne pakowanie w obiekty
    for(Integer i : c)
      System.out.print(i + ", ");
  }
} /* Output:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
*///:~
```

Ponieważ ten przykład korzysta jedynie z metod interfejsu `Collection`, zadziała dla dowolnego obiektu klasy dziedziczącej po `Collection`; `ArrayList` jest zaś najbardziej podstawowym typem sekwencji.

Metoda `add()`, jak sugeruje jej nazwa, zamieszcza nowy element w kolekcji. Jednak, jak dokładnie opisuje to dokumentacja, metoda ta „gwarantuje obecność elementu w kontenerze". To ukłon w stronę zbioru `Set`, który dodaje element tylko wtedy, jeśli jeszcze go tam nie ma. W przypadku `ArrayList` lub dowolnego rodzaju listy `add()` zawsze znaczy: „Umieść go wewnątrz", gdyż `List`y nic przejmują się powielaniami elementów.

Wszystkie kontenery typu `Collection` można przeglądać za pośrednictwem składni foreach. W dalszej części rozdziału poznamy bardziej elastyczną metodę przeglądania kontenerów — *iteratory*.

Ćwiczenie 2. Zmodyfikuj plik *SimpleCollection.java* tak, aby w roli klasy kontenerowej wystąpiła klasa `Set` (1).

Ćwiczenie 3. Zmodyfikuj plik *innerclasses/Sequence.java* tak, aby do sekwencji można było dodawać dowolną liczbę elementów (2).

Dodawanie grup elementów

Klasy Arrays i Collections z biblioteki java.util udostępniają metody narzędziowe grupujące elementy do postaci kolekcji Collection. Metoda Arrays.asList() przyjmuje tablicę albo listę elementów wymienianych po przecinku (za pomocą zmiennych list argumentów) i zwraca obiekt List. Z kolei Collections.addAll() przyjmuje obiekt Collection oraz tablicę bądź listę elementów i dodaje zawartość tablicy bądź komplet elementów do wskazanego obiektu Collection. Poniższy przykład ilustruje obie metody, a także bardziej konwencjonalną metodę addAll(), będącą na wyposażeniu wszystkich (pod)typów Collection:

```
//: holding/AddingGroups.java
// Dodawanie grup elementów do obiektów Collection.
import java.util.*;

public class AddingGroups {
  public static void main(String[] args) {
    Collection<Integer> collection =
      new ArrayList<Integer>(Arrays.asList(1, 2, 3, 4, 5));
    Integer[] moreInts = { 6, 7, 8, 9, 10 };
    collection.addAll(Arrays.asList(moreInts));
    // Działa znacznie szybciej, ale nie pozwala na zmontowanie
    // kolekcji Collection (jedynie na jej uzupełnienie):
    Collections.addAll(collection, 11, 12, 13, 14, 15);
    Collections.addAll(collection, moreInts);
    // Generuje listę z tablicą w roli "zaplecza":
    List<Integer> list = Arrays.asList(16, 17, 18, 19, 20);
    list.set(1, 99); // OK -- modyfikacja elementu
    // list.add(21); // Błąd czasu wykonania -- rozmiar
                     // tablicy zaplecza jest niezmienny.
  }
} ///:~
```

Konstruktor klas implementujących Collection może przyjmować obiekt Collection, który wykorzystuje do inicjalizacji nowego obiektu, co pozwala na przygotowywanie danych dla konstruktora za pomocą wywołania Arrays.asList(). Jednakże metoda Collections.addAll() działa znacznie szybciej i po prostu łatwiej jest skonstruować pusty obiekt Collection, aby potem wypełnić go szybkim wywołaniem Collections.addAll().

Metoda Collection.addAll() może przyjmować jedynie argument typu Collection, nie jest więc tak elastyczna jak Arrays.asList() czy Collections.adAll(), które mogą operować również na listach elementów przekazywanych za pomocą listy argumentów o zmiennej długości.

Można też korzystać wprost z wartości zwracanej przez Arrays.AsList(), traktując ją jako obiekt typu List, ale trzeba się liczyć z tym, że fundament („zaplecze") tak otrzymanej listy stanowi zwykła tablica, której rozmiar jest ustalony i niezmienny. Jeśli taką listę będziesz uzupełniał (add()) albo usuwał z niej elementy (delete()), spowodujesz próbę zmiany rozmiaru, a tym samym błąd czasu wykonania — wyjątek „operacji nieobsługiwanej".

Ograniczenie metody `Arrays.asList()` przejawia się również tym, że zakłada ona, że typem wynikowym ma być `List`, ale nie zwraca uwagi na to, co zostanie przekazane argumentami wywołania. Niekiedy może to sprawiać kłopot:

```
//: holding/AsListInference.java
// Arrays.asList() zgaduje typ.
import java.util.*;

class Snow {}
class Powder extends Snow {}
class Light extends Powder {}
class Heavy extends Powder {}
class Crusty extends Snow {}
class Slush extends Snow {}

public class AsListInference {
  public static void main(String[] args) {
    List<Snow> snow1 = Arrays.asList(
      new Crusty(), new Slush(), new Powder());

    // Nie da się skompilować:
    // List<Snow> snow2 = Arrays.asList(
    //   new Light(), new Heavy());
    // Kompilator stwierdzi:
    // found   : java.util.List<Powder>
    // required: java.util.List<Snow>

    // Collections.addAll() nie ma z tym problemu:
    List<Snow> snow3 = new ArrayList<Snow>();
    Collections.addAll(snow3, new Light(), new Heavy());

    // Pomożemy, jawnie podpowiadając typ argumentu:
    List<Snow> snow4 = Arrays.<Snow>asList(
      new Light(), new Heavy());
  }
} ///:~
```

Przy próbie utworzenia obiektu `snow2` metoda `Arrays.asList()` rozpoznaje podtypy `Powder`, więc zwraca `List<Powder>` zamiast oczekiwanego `List<Snow>`; tymczasem `Collections.addAll()` radzi sobie lepiej, bo pierwszy argument wskazuje jej dokładny typ docelowy.

Przypadek tworzenia obiektu `snow4` ujawnia, że w wywołaniu metody `Arrays.asList()` można umieścić „podpowiedź", wskazówkę co do właściwego typu docelowego listy. To tak zwana *jawna specyfikacja argumentu typowego*.

Odwzorowania `Map` są bardziej złożone, a biblioteka standardowa Javy nie przewiduje automatycznej inicjalizacji takich kontenerów z wyjątkiem inicjalizacji zawartością już istniejącego kontenera `Map`.

Wypisywanie zawartości kontenerów

Aby wygenerować dającą się wypisać reprezentację tablicy, trzeba skorzystać z pośrednictwa metody `Arrays.toString()`; tymczasem kontenery nie wymagają żadnych dodatkowych operacji. Poniższy przykład pokazuje, jak to zrobić, a przy okazji wprowadzi Cię w podstawowe rodzaje kontenerów:

```
//: holding/PrintingContainers.java
// Kontenery automatycznie wypisują swoją zawartość.
import java.util.*;
import static net.mindview.util.Print.*;

public class PrintingContainers {
  static Collection fill(Collection<String> collection) {
    collection.add("szczur");
    collection.add("kot");
    collection.add("pies");
    collection.add("pies");
    return collection;
  }
  static Map fill(Map<String,String> map) {
    map.put("szczur", "Gonek");
    map.put("kot", "Maja");
    map.put("pies", "Bosco");
    map.put("pies", "Spot");
    return map;
  }
  public static void main(String[] args) {
    print(fill(new ArrayList<String>()));
    print(fill(new LinkedList<String>()));
    print(fill(new HashSet<String>()));
    print(fill(new TreeSet<String>()));
    print(fill(new LinkedHashSet<String>()));
    print(fill(new HashMap<String,String>()));
    print(fill(new TreeMap<String,String>()));
    print(fill(new LinkedHashMap<String,String>()));
  }
} /* Output:
[szczur, kot, pies, pies]
[szczur, kot, pies, pies]
[pies, szczur, kot]
[kot, pies, szczur]
[szczur, kot, pies]
{pies=Spot, szczur=Gonek, kot=Maja}
{kot=Maja, pies=Spot, szczur=Gonek}
{szczur=Gonek, kot=Maja, pies=Spot}
*///:~
```

Mamy tu dwie podstawowe kategorie w bibliotece kontenerów Javy. Różnica polega na innej liczbie elementów zamieszczonych na każdej z pozycji. Kontener z kategorii `Collection` przechowuje tylko pojedyncze elementy. Do kategorii tej zalicza się lista reprezentowana przez `List`, która przechowuje grupę elementów w określonej kolejności, oraz zbiór `Set`, który pozwala na dodanie tylko pojedynczego elementu danego rodzaju, czy kolejka `Queue`, która pozwala wstawiać obiekty jedynie na „koniec" kontenera i usuwać

je z jego drugiego „końca" (w ostatnim przykładzie nie mieliśmy ilustracji kolejki). Z kolei odwzorowanie Map przechowuje na swoich pozycjach nierozerwalne pary obiektów: *klucze* skojarzone z *wartościami*.

Jeśli spojrzymy na wyjście programu, to zauważymy, że domyślne wypisywanie (dostępne przez różne metody toString() kontenera) daje dosyć czytelny wynik. Elementy Collection są oddzielone przecinkami i wypisywane w nawiasie kwadratowym, a elementy Map w nawiasie klamrowym — jako klucz i skojarzona wartość, rozdzielone znakiem równości (klucze po lewej, wartości po prawej).

Pierwsza metoda fill() działa z wszystkimi typami Collection, z których każdy implementuje metodę add() realizującą wstawianie nowych elementów do kolekcji.

Kolekcje ArrayList i LinkedList to podtypy List; na wyjściu programu widać, że obie kolekcje przechowują dodawane obiekty w kolejności zgodnej z kolejnością dodawania. Różnice pomiędzy nimi dotyczą nie tylko wydajności poszczególnych operacji na kolekcjach — otóż klasa LinkedList udostępnia więcej operacji niż ArrayList. Różnicami tymi zajmiemy się w dalszej części rozdziału.

Z kolei klasy zbiorów: HashSet, TreeSet i LinkedHashSet to podtypy Set. Wyjście programu ujawnia, że zbiór może przechowywać tylko jeden element o danej wartości, pokazuje też różnice odnośnie przechowywania elementów w różnych implementacjach Set. Otóż HashSet przy przechowywaniu elementów korzysta z raczej wyszukanej techniki, wyjaśnionej w rozdziale „Kontenery z bliska" — na razie wystarczy wiedzieć, że technika ta pozwala na bardzo szybkie wybieranie elementów z kontenera, za to kolejność przechowywania elementów zdaje się zupełnie nonsensowna (ale zbiory mają to do siebie, że interesuje nas ich skład, a nie ułożenie poszczególnych elementów względem siebie). Jeśli zachowanie kolejności, w jakiej elementy były dodawane, ma znaczenie, można wykorzystać klasę TreeSet (która przechowuje elementy według wartości) albo LinkedHashSet (która zachowuje kolejność wstawiania elementów).

Odwzorowanie Map (zwane też *tablicą asocjacyjną*) pozwala na wybieranie obiektów na podstawie *kluczy*, jak w prostej bazie danych. Wydobyty tak obiekt jest *wartością* skojarzoną z kluczem. Gdybyśmy na przykład posiadali kontener odwzorowujący nazwy stanów na nazwy ich stolic, to w celu poznania stolicy Ohio zainicjowalibyśmy wydobycie elementu według klucza Ohio — prawie tak, jakby „Ohio" było indeksem tablicy nazw stolic. Ta specyfika oznacza niemożność dublowania kluczy.

Metoda Map.put(klucz, wartość) dodaje do kontenera wartość (to, co chcemy przechowywać) i kojarzy ją z kluczem (tym, według czego chcemy wyszukiwać). Metoda Map.get(klucz) zwraca wartość skojarzoną z przekazanym kluczem. W powyższym przykładzie jedynie dodawaliśmy pary klucz-wartość, nie wypróbowaliśmy zaś wyszukiwania — będzie na to czas.

Zauważ, że nie trzeba nigdzie podawać rozmiaru odwzorowania Map (ani nawet troszczyć się o ten rozmiar), bo rozmiar kontenera automatycznie dopasowuje się do potrzeb. Do tego Map wie, jak wypisać swoją zawartość, ukazując skojarzenia kluczy z wartościami. Widoczna na wyjściu kolejność przechowywania par nie jest tożsama z kolejnością ich wstawiania, bo implementacja HashMap opiera się na wspomnianym algorytmie szybkiego dostępu do wartości kosztem zaburzenia kolejności elementów.

Przykład operuje na trzech podstawowych odmianach odwzorowania: HashMap, TreeMap i LinkedHashMap. Tak jak w przypadku kolekcji HashSet, HashMap zapewnia najszybsze wyszukiwanie elementów, za to nie zachowuje ich pierwotnej kolejności. TreeMap porządkuje elementy według wartości (rosnąco), a LinkedHashMap zachowuje kolejność wstawiania elementów przy równoczesnym zapewnieniu szybkości wyszukiwania porównywalnej z HashMap.

Ćwiczenie 4. Utwórz klasę *generatora*, która za każdym kolejnym wywołaniem metody next() będzie zwracać nazwy (obiekty String) postaci z Twojego ulubionego filmu (jeśli nic nie przyjdzie Ci do głowy, zawsze możesz wybrać *Gwiezdne Wojny* albo *Królewnę Śnieżkę*); po wyczerpaniu listy postaci metoda zacznie przeglądać listę postaci od początku. Wykorzystaj generator do wypełnienia tablicy kontenerów ArrayList, LinkedList, HashSet, LinkedHashSet i TreeSet; wypisz zawartość każdego z kontenerów (3).

Interfejs List

List obiecuje zachowanie kolejności elementów. Interfejs uzupełnia interfejs Collection o zestaw metod pozwalających na wstawianie i usuwanie elementów do i ze środka kolekcji.

Wyróżniamy dwa podtypy List:

- Podstawowy, ArrayList, celujący w operacjach swobodnego dostępu do elementów, ale wolniejszy przy wstawianiu i usuwaniu elementów z listy.
- LinkedList, optymalny dla dostępu sekwencyjnego, z efektywnymi operacjami wstawiania elementów (i ich usuwania) do środka listy. Operacje dostępu swobodnego LinkedList realizuje stosunkowo wolno, ale za to udostępnia bogatszy zestaw funkcji.

Kolejny przykład będzie wybiegał naprzód, odwołując się przez import typeinfo.pets do biblioteki z rozdziału „Informacje o typie". Biblioteka ta zawiera hierarchię klas Pet (zwierzątko) z kompletem narzędzi do losowego generowania obiektów klas hierarchii. Szczegóły jej implementacji są w tej chwili mało istotne, wystarczy wiedzieć, że (1) istnieje klasa Pet z zestawem podtypów i że (2) statyczna metoda Pets.arrayList() zwraca kolekcję ArrayList wypełnioną wylosowanymi obiektami zwierzaków.

```
//: holding/ListFeatures.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ListFeatures {
  public static void main(String[] args) {
    Random rand = new Random(47);
    List<Pet> pets = Pets.arrayList(7);
    print("1: " + pets);
    Hamster h = new Hamster();
    pets.add(h); // Automatyczne dopasowanie rozmiaru
    print("2: " + pets);
    print("3: " + pets.contains(h));
    pets.remove(h); // Usuwanie przez obiekt
    Pet p = pets.get(2);
```

```
    print("4: " +  p + " " + pets.indexOf(p));
    Pet cymric = new Cymric();
    print("5: " + pets.indexOf(cymric));
    print("6: " + pets.remove(cymric));
    // Potrzebny dokładnie ten egzemplarz:
    print("7: " + pets.remove(p));
    print("8: " + pets);
    pets.add(3, new Mouse()); // Wstawienie pod indeks
    print("9: " + pets);
    List<Pet> sub = pets.subList(1, 4);
    print("subList: " + sub);
    print("10: " + pets.containsAll(sub));
    Collections.sort(sub); // Sortowanie miejscowe
    print("posortowana lista subList: " + sub);
    // W containsAll() kolejność jest nieistotna:
    print("11: " + pets.containsAll(sub));
    Collections.shuffle(sub, rand); // Wymieszanie
    print("wymieszana lista subList: " + sub);
    print("12: " + pets.containsAll(sub));
    List<Pet> copy = new ArrayList<Pet>(pets);
    sub = Arrays.asList(pets.get(1), pets.get(4));
    print("sub: " + sub);
    copy.retainAll(sub);
    print("13: " + copy);
    copy = new ArrayList<Pet>(pets); // Świeża kopia
    copy.remove(2); // Usuwanie przez indeks
    print("14: " + copy);
    copy.removeAll(sub); // Usuwanie jedynie dokładnie
                         // takich egzemplarzy
    print("15: " + copy);
    copy.set(1, new Mouse()); // Zastąpienie elementu
    print("16: " + copy);
    copy.addAll(2, sub); // Wstawienie listy w środek
    print("17: " + copy);
    print("18: " + pets.isEmpty());
    pets.clear(); // Usunięcie wszystkich elementów
    print("19: " + pets);
    print("20: " + pets.isEmpty());
    pets.addAll(Pets.arrayList(4));
    print("21: " + pets);
    Object[] o = pets.toArray();
    print("22: " + o[3]);
    Pet[] pa = pets.toArray(new Pet[0]);
    print("23: " + pa[3].id());
  }
} /* Output:
1: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug]
2: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Hamster]
3: true
4: Cymric 2
5: -1
6: false
7: true
8: [Rat, Manx, Mutt, Pug, Cymric, Pug]
9: [Rat, Manx, Mutt, Mouse, Pug, Cymric, Pug]
subList: [Manx, Mutt, Mouse]
10: true
posortowana lista subList: [Manx, Mouse, Mutt]
```

```
11: true
wymieszana lista subList: [Mouse, Manx, Mutt]
12: true
sub: [Mouse, Pug]
13: [Mouse, Pug]
14: [Rat, Mouse, Mutt, Pug, Cymric, Pug]
15: [Rat, Mutt, Cymric, Pug]
16: [Rat, Mouse, Cymric, Pug]
17: [Rat, Mouse, Mouse, Pug, Cymric, Pug]
18: false
19: []
20: true
21: [Manx, Cymric, Rat, EgyptianMau]
22: EgyptianMau
23: 14
*///:~
```

Wiersze wyjścia zostały ponumerowane, żeby nie było wątpliwości, z których wierszy kodu pochodzą. Pierwszy wiersz wyjścia prezentuje pierwotną listę zwierzaków. W przeciwieństwie do zwykłej tablicy lista pozwala na dodawanie elementów po utworzeniu, a także usuwanie elementów, przy automatycznym dopasowaniu rozmiaru. Modyfikowalność sekwencji elementów to zasadnicza zaleta. Drugi wiersz prezentuje efekt dodania elementu `Hamster` (chomik) — nowy element wylądował na końcu listy.

Obecność obiektu na liście sprawdza się za pomocą metody `contains()`. Aby usunąć obiekt, należy przekazać referencję tego obiektu do metody `remove()`. Ponadto posiadając referencję obiektu, można pozyskać numer indeksu, pod którym ten obiekt występuje na liście `List` — służy do tego metoda `indexOf()`, której efekt widać w czwartym wierszu wyjścia.

Przy sprawdzaniu, czy obiekt występuje na liście, pozyskiwaniu indeksu elementu listy i usuwaniu obiektu na podstawie referencji implementacja `List` odwołuje się do metody `equals()` (z klasy bazowej `Object`). Każdy obiekt `Pet` jest definiowany jako unikatowy egzemplarz, więc mimo że lista zawiera dwa koty walijskie (egzemplarze klasy `Cymric`), jeśli utworzyć nowy obiekt klasy `Cymric` i sprawdzić jego indeks wywołaniem `indexOf()`, otrzyma się w wyniku `-1` (brak elementu na liście); podobnie próba usunięcia obiektu wywołaniem `remove()` zwróci wartość `false`. W przypadku innych klas metoda `equals()`, wykorzystywana do porównywania obiektów, może być zdefiniowana inaczej — na przykład jej wersja dla klasy `String` porównuje zawartość ciągów reprezentowanych egzemplarzami klasy: obiekty identycznych ciągów są uznawane za identyczne. Aby uniknąć niespodzianek, należałoby więc pamiętać, że zachowanie listy `List` zależy w pewnej mierze od zachowania metody `equals()` dla typu elementów listy.

Wiersze wyjściowe o numerach 7 i 8 sygnalizują usuwanie obiektów pasujących (dokładnie) do referencji przekazanej w wywołaniu — usuwanie jest w takim przypadku skuteczne.

Program pokazuje też możliwość wstawiania elementów do środka sekwencji `List`, której dowodzi 9. wiersz wyjścia; tu pojawia się jednak kwestia wątpliwa: dla podtypu `LinkedList` operacje wstawiania i usuwania w środku listy to operacje tanie (kosztowne jest jedynie samo swobodne odwołanie do elementu z wnętrza listy), ale dla implementacji `ArrayList` jest to operacja dość kosztowna. Czy to oznacza, że listy `ArrayList` nie nadają się w ogóle do wstawiania elementów na środkowe pozycje i czy należałoby

wtedy bezwarunkowo korzystać z LinkedList? Niekoniecznie, znaczy to jedynie, że trzeba być świadomym ewentualnego zwiększonego kosztu: jeśli zaczniemy intensyfikować operacje wstawiania do środka ArrayList *i jednocześnie* zaobserwujemy spowolnienie programu, możemy podejrzewać o nie właśnie implementację listy (sposoby wykrywania wąskich gardeł znajdziesz w suplemencie publikowanym pod adresem *http://MindView.net/Books/BetterJava* — w skrócie, chodzi o program profilujący). Optymalizacja to zawsze ryzykowna sprawa i najlepiej odkładać ją do momentu, w którym naprawdę jest potrzebna (co nie znaczy, że można zupełnie lekceważyć kwestie wydajności poszczególnych operacji).

Metoda subList() pozwala na łatwe wyodrębnianie mniejszych list w większych; wynik wywołania tej metody w roli argumentu metody containsAll() wywołanej na rzecz listy źródłowej w naturalny sposób wymusza zwrócenie wartości true. Co ciekawe, kolejność elementów na liście jest wtedy nieistotna — można to zaobserwować w wierszach wyjściowych 11. i 12., gdzie podlista sub została przetworzona metodami Collections.sort() (porządkowanie kolekcji) i Collections.shuffle() (tasowanie kolekcji) — zmiana uporządkowania nie wpłynęła na wynik containsAll(). Metoda subList() zwraca niejako częściową perspektywę listy oryginalnej — zmiany listy wyodrębnionej są odzwierciedlane w liście oryginalnej i na odwrót.

Metoda retainAll() to operacja „części wspólnej", charakterystycznej dla zbiorów; w tym przypadku zachowuje wszystkie elementy z copy, które występują również na liście sub. Zachowanie tej metody, jako porównującej elementy listy, zależy od definicji metody equals().

14. wiersz wyjścia ilustruje efekt usuwania elementu na podstawie jego indeksu, co jest o tyle prostsze niż usuwanie na bazie referencji, że nie trzeba się martwić o wpływ metody equals() na dopasowanie elementu do usunięcia.

Metoda removeAll() również bazuje na wynikach equals(). Zgodnie ze swoją nazwą usuwa wszystkie te obiekty listy, na rzecz której zostanie wywołana, które występują również na liście przekazanej argumentem.

Nazwa dla metody set() jest dobrana nieco niefortunnie, bo może mylić się z klasą Set — lepszą nazwą byłoby pewnie „replace" (zastąp), bo też działanie metody sprowadza się do zastąpienia elementu spod wskazanego indeksu (pierwszy argument wywołania) obiektem przekazanym przez drugi argument wywołania metody.

Wiersz numer 17 pokazuje, że listy List posiadają przeciążoną metodę addAll(), którą można wykorzystać do wstawienia całych list w środek pierwotnej listy — dla przypomnienia, addAll() z klasy Collection dodaje listę elementów na koniec listy docelowej.

Wiersze wyjściowe o numerach od 18 do 20 prezentują efekty wywołań metod isEmpty() i clear().

Wiersze 22. i 23. demonstrują możliwość konwersji dowolnej kolekcji Collection na postać tablicy za pomocą metody toArray(). To metoda przeciążona; wersja bezargumentowa zwraca tablicę obiektów klasy Object, ale jeśli do wersji przeciążonej przekażemy tablicę typu docelowego, metoda wygeneruje tablicę obiektów odpowiedniego typu (o ile typy będą zgodne). Jeśli przekazana tablica nie mieści wszystkich obiektów z listy

(jak na przykładzie), `toArray()` tworzy nową tablicę o odpowiednim rozmiarze. W przykładzie wynikowa tablica jest przeglądana, a dla każdego elementu wywoływana jest metoda `id()` klasy `Pet`.

Ćwiczenie 5. Zmodyfikuj plik *ListFeatures.java* tak, aby zamiast obiektów `Pet` lista zawierała obiekty `Integer` (pamiętaj o automatycznym pakowaniu wartości typów podstawowych w obiekty!); wyjaśnij różnice w wynikach (3).

Ćwiczenie 6. Zmodyfikuj plik *ListFeatures.java* tak, aby zamiast obiektów `Pet` lista zawierała obiekty `String`; wyjaśnij różnice w wynikach (2).

Ćwiczenie 7. Utwórz klasę, a potem zainicjalizowaną tablicę obiektów tej klasy. Zawartością tablicy wypełnij listę `List`. Wyłuskaj z niej fragment listy metodą `subList()`, a następnie usuń tę podlistę z oryginalnej listy (3).

Interfejs Iterator

W każdej klasie kontenerowej potrzebny jest nam sposób na zamieszczanie elementów oraz sposób na ich pobieranie. W końcu jest to podstawowe działanie kontenera — przechowywanie różnych rzeczy. W przypadku `List` jednym ze sposobów dodawania elementów jest metoda `add()`, a jednym ze sposobów ich wydobywania — metoda `get()`.

Jeżeli chcesz zacząć pracować na nieco wyższym poziomie, musisz wiedzieć o pewnej wadzie: aby użyć kontenera, trzeba znać jego dokładny typ. Początkowo może się to nie wydawać złe, ale co jeśli rozpoczniemy od `List`, a później w programie okaże się, iż ze względu na sposób wykorzystywania kontenera znacznie bardziej wydajne byłoby użycie `LinkedList`. Albo przypuśćmy, że chcielibyśmy napisać kawałek kodu, który nie wie lub nie dba o to, z jakiego typu kontenerem ma do czynienia, tak by mógł być zastosowany dla różnorodnych kontenerów bez potrzeby przepisywania kodu.

Do uzyskania takiej abstrakcji może być wykorzystane pojęcie *iteratora* (będącego kolejnym wzorcem projektowym). Iterator to obiekt, którego zadaniem jest przemieszczanie się po sekwencji elementów i wybieranie każdego z napotkanych obiektów bez wiedzy programisty-użytkownika lub przejmowania się wewnętrzną strukturą takiej sekwencji. Dodatkowo iterator jest *„lekkim" obiektem* — jego stworzenie jest mało kosztowne. Z tego powodu często napotkamy pozornie dziwne ograniczenia iteratorów, na przykład iterator `Iterator` Javy może przesuwać się tylko w jednym kierunku. Można z nim zrobić niewiele:

1. Poprosić kolekcję `Collection` o udostępnienie `Iteratora`, wywołując jej metodę o nazwie `iterator()`. Iterator ten będzie gotów do zwrócenia pierwszego elementu sekwencji.
2. Uzyskać następny obiekt z ciągu dzięki metodzie `next()`.
3. Sprawdzić, czy są jakieś inne obiekty dalej w sekwencji za pomocą `hasNext()`.
4. Usunąć ostatni zwrócony przez iterator element, stosując `remove()`.

Aby zobaczyć, jak to działa, wykorzystamy ponownie bibliotekę zwierzaków z rozdziału „Informacje o typach":

```
// : holding/SimpleIteration.java
import typeinfo.pets.*;
import java.util.*;

public class SimpleIteration {
  public static void main(String[] args) {
    List<Pet> pets = Pets.arrayList(12);
    Iterator<Pet> it = pets.iterator();
    while(it.hasNext()) {
      Pet p = it.next();
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
    // Wersja prostsza, jeśli możliwa:
    for(Pet p : pets)
      System.out.print(p.id() + ":" + p + " ");
    System.out.println();
    // Iterator może również usuwać elementy:
    it = pets.iterator();
    for(int i = 0; i < 6; i++) {
      it.next();
      it.remove();
    }
    System.out.println(pets);
  }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx 8:Cymric 9:Rat 10:EgyptianMau
11:Hamster
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx 8:Cymric 9:Rat 10:EgyptianMau
11:Hamster
[Pug, Manx, Cymric, Rat, EgyptianMau, Hamster]
*///:~
```

Dzięki iteratorowi nie trzeba się kłopotać numerami elementów kontenera — wszystko załatwiają wywołania metod `hasNext()` i `next()`.

W przypadku prostego przeglądania listy od początku do końca bez podejmowania prób modyfikowania elementów listy najlepiej skorzystać po prostu ze składni foreach.

Iterator może też usunąć element zwrócony ostatnim wywołaniem metody `next()`, co oznacza, że każde wywołanie metody `remove()` musi być poprzedzone wywołaniem `next()`[4].

Koncepcja przyjmowania kontenera obiektów i przekazywania go w celu wykonania operacji na każdym z jego elementów będzie wykorzystywana jeszcze wielokrotnie w tej książce jako jeden z efektywnych idiomów programistycznych.

[4] Metoda `remove()` jest tak zwaną (jedną z wielu) metodą „opcjonalną", co oznacza, że nie wszystkie implementacje interfejsu Iterator muszą ją implementować. Kwestia metod opcjonalnych zostanie poruszona ponownie w rozdziale „Kontenery z bliska". Jednak kontenery biblioteki standardowej języka Java bez wyjątku implementują metodę `remove` dla interfejsu `Iterator`, więc przynajmniej do końca tego rozdziału nie musimy się tym martwić.

Jako kolejny przykład rozważmy stworzenie uniwersalnej (niezależnej od kontenera) metody wypisującej elementy:

```
//: holding/CrossContainerIteration.java
import typeinfo.pets.*;
import java.util.*;

public class CrossContainerIteration {
  public static void display(Iterator<Pet> it) {
    while(it.hasNext()) {
      Pet p = it.next();
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
  }
  public static void main(String[] args) {
    ArrayList<Pet> pets = Pets.arrayList(8);
    LinkedList<Pet> petsLL = new LinkedList<Pet>(pets);
    HashSet<Pet> petsHS = new HashSet<Pet>(pets);
    TreeSet<Pet> petsTS = new TreeSet<Pet>(pets);
    display(pets.iterator());
    display(petsLL.iterator());
    display(petsHS.iterator());
    display(petsTS.iterator());
  }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
5:Cymric 2:Cymric 7:Manx 1:Manx 3:Mutt 6:Pug 4:Pug 0:Rat
*///:~
```

Zauważ, że metoda `display()` nie posiada żadnej wiedzy o typie sekwencji, którą przegląda, co ilustruje największą zaletę iteratorów: zdolność do oddzielenia operacji przeglądania sekwencji od wewnętrznej struktury tejże sekwencji. Właśnie dlatego mówi się niekiedy, że iteratory unifikują dostęp do kontenerów.

Ćwiczenie 8. Zmodyfikuj ćwiczenie 1. tak, aby przeglądało listę (i wywoływało metodę `hop()` elementów listy) za pomocą iteratora (1).

Ćwiczenie 9. Zmień plik *innerclasses/Sequence.java* tak, aby sekwencja `Sequence` operowała iteratorem, a nie selektorem `Selector` (4).

Ćwiczenie 10. Zmień ćwiczenie 9. z rozdziału „Polimorfizm" tak, aby lista `Gryzoni` była przechowywana jako `ArrayList`, a do jej przeglądania służył iterator (2).

Ćwiczenie 11. Napisz metodę, która wykorzysta iterator do przejrzenia kontenera `Collection` i wypisze wynik wywołania `toString()` na rzecz każdego z obiektów kontenera. Utwórz i wypełnij rozmaite kontenery `Collection` obiektami i do każdego z nich zaaplikuj napisaną metodę (2).

Interfejs ListIterator

`ListIterator` to nieco rozbudowany podtyp interfejsu `Iterator`, zwracany jedynie przez klasy `List`. Gdy najzwyklejszy `Iterator` może być przesuwany jedynie do przodu, `ListIterator` jest iteratorem dwukierunkowym. Może też zwracać indeksy elementu poprzedniego i następnego, względem bieżącej pozycji iteratora na liście, a także pozwala na zastępowanie ostatnio odwiedzonego elementu za pomocą metody `set()`. `ListIterator` wskazujący początek listy `List` tworzy się wywołaniem metody `listIterator()`; można także utworzyć iterator `ListIterator` ustawiony na element o indeksie `n` — wystarczy wywołać `listIterator(n)`. Oto przykład pokazujący wszystkie możliwości tego iteratora:

```
//: holding/ListIteration.java
import typeinfo.pets.*;
import java.util.*;

public class ListIteration {
  public static void main(String[] args) {
    List<Pet> pets = Pets.arrayList(8);
    ListIterator<Pet> it = pets.listIterator();
    while(it.hasNext())
      System.out.print(it.next() + ", " + it.nextIndex() +
        ", " + it.previousIndex() + "; ");
    System.out.println();
    // Wstecz:
    while(it.hasPrevious())
      System.out.print(it.previous().id() + " ");
    System.out.println();
    System.out.println(pets);
    it = pets.listIterator(3);
    while(it.hasNext()) {
      it.next();
      it.set(Pets.randomPet());
    }
    System.out.println(pets);
  }
} /* Output:
Rat, 1, 0; Manx, 2, 1; Cymric, 3, 2; Mutt, 4, 3; Pug, 5, 4; Cymric, 6, 5; Pug, 7, 6; Manx, 8, 7;
7 6 5 4 3 2 1 0
[Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Manx]
[Rat, Manx, Cymric, Cymric, Rat, EgyptianMau, Hamster, EgyptianMau]
*///:~
```

Widać tu zastosowanie metody `Pets.randomPet()` do zastępowania obiektów `Pet` listy `List` od trzeciej pozycji do końca.

Ćwiczenie 12. Utwórz i zapełnij listę `List<Integer>`. Utwórz drugą listę `List<Integer>` tego samego rozmiaru co pierwsza i użyj iteratora `ListIterator` do przejrzenia elementów pierwszej listy i wstawienia ich do listy drugiej, ale w odwrotnej kolejności (spróbuj wykonać zadanie na kilka sposobów) (3).

Klasa LinkedList

Klasa `LinkedList` również — tak jak `ArrayList` — implementuje podstawowy interfejs `List`, ale realizuje pewne operacje dodatkowe (wstawianie elementów do środka listy i usuwanie ich stamtąd) znacznie efektywniej niż `ArrayList`. Jest za to mniej wydajna w realizacji swobodnego dostępu do elementów listy.

Klasa `LinkedList` udostępnia też metody, które pozwalają na wykorzystanie jej w roli stosu, kolejki albo kolejki dwukierunkowej (ang. *deque*).

Niektóre z tych metod są aliasami albo odmianami, co ma pozwalać na różnicowanie nazw w zależności od kontekstu użycia danej operacji. Na przykład metody `getFirst()` i `element()` są identyczne — obie zwracają czołowy (pierwszy) element listy, nie usuwając go z niej i ewentualnie zgłaszając wyjątek `NoSuchElementException`, jeśli lista jest całkiem pusta. Z kolei metoda `peek()` to jedynie niewielka wariacja na temat dwóch poprzednich: kiedy lista jest pusta, nie zgłasza wyjątku, a jedynie zwraca `null`.

Identycznie zachowują się też metody `removeFirst()` i `remove()` — obie usuwają z listy i zwracają czołowy element, a kiedy lista jest pusta, zgłaszają wyjątek `NoSuchElementException`; z kolei `poll()` to odmiana, która dla pustej listy zwraca wartość `null`.

Metoda `addFirst()` wstawia element na początek listy.

Metoda `offer()` działa jak `add()` i `addLast()` — wszystkie trzy dodają element na ogon (koniec) listy.

Metoda `removeLast()` usuwa i zwraca ostatni element listy.

Poniżej zamieszczony został przykład ilustrujący podstawowe funkcje i różnice pomiędzy nimi. Przykład nie powiela demonstracji zachowania, które prześledziliśmy już na przykładzie programu *ListFeatures.java*:

```
// holding/LinkedListFeatures.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class LinkedListFeatures {
  public static void main(String[] args) {
    LinkedList<Pet> pets =
      new LinkedList<Pet>(Pets.arrayList(5));
    print(pets);
    // Identyczne:
    print("pets.getFirst(): " + pets.getFirst());
    print("pets.element(): " + pets.element());
    // Różnią się jedynie zachowaniem w obliczu braku elementów:
    print("pets.peek(): " + pets.peek());
    // Identyczne; usuwają i zwracają pierwszy element listy:
    print("pets.remove(): " + pets.remove());
    print("pets.removeFirst(): " + pets.removeFirst());
    // Różnią się jedynie zachowaniem w obliczu braku elementów:
    print("pets.poll(): " + pets.poll());
```

```
        print(pets);
        pets.addFirst(new Rat());
        print("Po addFirst(): " + pets);
        pets.offer(Pets.randomPet());
        print("Po offer(): " + pets);
        pets.add(Pets.randomPet());
        print("Po add(): " + pets);
        pets.addLast(new Hamster());
        print("Po addLast(): " + pets);
        print("pets.removeLast(): " + pets.removeLast());
      }
    } /* Output:
    [Rat, Manx, Cymric, Mutt, Pug]
    pets.getFirst(): Rat
    pets.element(): Rat
    pets.peek(): Rat
    pets.remove(): Rat
    pets.removeFirst(): Manx
    pets.poll(): Cymric
    [Mutt, Pug]
    Po addFirst(): [Rat, Mutt, Pug]
    Po offer(): [Rat, Mutt, Pug, Cymric]
    Po add(): [Rat, Mutt, Pug, Cymric, Pug]
    Po addLast(): [Rat, Mutt, Pug, Cymric, Pug, Hamster]
    pets.removeLast(): Hamster
    *///:~
```

Wypełnienie listy `LinkedList` odbywa się przez przekazanie do konstruktora wyniku wywołania metody `Pets.arrayList()`. Jeśli spojrzysz na interfejs kolejki `Queue`, zobaczysz metody `element()`, `offer()`, `peek()`, `poll()` i `remove()` dodane do `LinkedList` po to, aby lista mogła występować w roli kolejki. Pełniejsze przykłady z kolejkami zostaną zaprezentowane w dalszej części rozdziału.

Ćwiczenie 13. Klasa `Controller` w pliku *innerclasses/GreenhouseController.java* korzysta z kontenera `ArrayList`. Zmień kod programu tak, aby klasa `Controller` wykorzystywała kontener `LinkedList`, a do przeglądania zdarzeń używała iteratora (3).

Ćwiczenie 14. Utwórz pusty kontener `LinkedList<Integer>`. Za pomocą iteratora `ListIterator` zapełnij kontener elementami, wstawiając je zawsze na środek listy (3).

Klasa Stack

Stos (ang. *stack*) jest niekiedy przedstawiany jako kontener typu *last-in, first-out* (LIFO). Cokolwiek zostanie *włożone na stos* (ang. *push*) jako *ostatnie*, jest pierwszym elementem, który można z niego zdjąć (ang. *pop*). Pasuje tu analogia do podajnika tac w kawiarni — kelnerka zawsze odbiera tę tacę, która została wsunięta jako ostatnia.

`LinkedList` zawiera metody, które bezpośrednio wprowadzają funkcje stosu, zatem można również użyć tej klasy, zamiast tworzyć klasę stosu. Jednakże czasami klasa stosu może lepiej odzwierciedlać sytuację:

```
//: net/mindview/util/Stack.java
// Stos na bazie LinkedList.
package net.mindview.util;
import java.util.LinkedList;

public class Stack<T> {
  private LinkedList<T> storage = new LinkedList<T>();
  public void push(T v) { storage.addFirst(v); }
  public T peek() { return storage.getFirst(); }
  public T pop() { return storage.removeFirst(); }
  public boolean empty() { return storage.isEmpty(); }
  public String toString() { return storage.toString(); }
} ///:~
```

Mamy tu najprostszy z możliwych przykładów definicji klasy z użyciem typu ogólnego. Otóż <T> za nazwą klasy to wskazówka dla kompilatora, że klasa będzie *typem parametryzowanym* z parametrem typowym — który będzie w miejscu użycia klasy zastępowany właściwą nazwą typu — pod nazwą T. Zapis taki mówi, że „definiujemy stos Stack przechowujący obiekty typu T". Sam stos jest implementowany na bazie kontenera LinkedList, przy czym obiekt LinkedList również występuje jako kontener obiektów typu T. Zauważ, że metoda push() przyjmuje w wywołaniu obiekt typu T, a metody peek() i pop() zwracają obiekt T. Metoda peek() pozwala na podejrzenie wartości szczytowego elementu bez zdejmowania go ze stosu; pop() zwraca zdjęty szczytowy element stosu.

Jeśli chcemy uzyskać jedynie zachowanie stosu, to mechanizm dziedziczenia jest tu nieodpowiedni, ponieważ dostalibyśmy klasę z całą resztą metod klasy LinkedList (w rozdziale „Kontenery z bliska" przekonasz się, że ten właśnie błąd popełnili projektanci klasy Stack w bibliotece java.util.Stack).

Oto prosta demonstracja nowej klasy stosu — Stack:

```
//: holding/StackTest.java
import net.mindview.util.*;

public class StackTest {
  public static void main(String[] args) {
    Stack<String> stack = new Stack<String>();
    for(String s : "Mój pies ma pchły".split(" "))
      stack.push(s);
    while(!stack.empty())
      System.out.print(stack.pop() + " ");
  }
} /* Output:
Pchły ma pies Mój
*///:~
```

Aby wykorzystać klasę Stack we własnym kodzie, należy przy tworzeniu obiektu tej klasy albo podać pełną nazwę pakietu, albo zmienić nazwę klasy; inaczej najprawdopodobniej dojdzie do kolizji z klasą Stack z pakietu java.util. Gdybyśmy na przykład w poprzednim przykładzie wykonali import w postaci import java.util.*, musielibyśmy zapobiegać kolizji przez kwalifikowanie nazwy klasy nazwą pakietu:

```
//: holding/StackCollision.java
import net.mindview.util.*;
```

```
public class StackCollision {
  public static void main(String[] args) {
    net.mindview.util.Stack<String> stack =
      new net.mindview.util.Stack<String>();
    for(String s : "Mój pies ma pchły".split(" "))
      stack.push(s);
    while(!stack.empty())
      System.out.print(stack.pop() + " ");
    System.out.println();
    java.util.Stack<String> stack2 =
      new java.util.Stack<String>();
    for(String s : "Mój pies ma pchły".split(" "))
      stack2.push(s);
    while(!stack2.empty())
      System.out.print(stack2.pop() + " ");
  }
} /* Output:
pchły ma pies Mój
pchły ma pies Mój
*///:~
```

Obie klasy Stack mają identyczne interfejsy, ale w pakiecie java.util nie istnieje wspólny interfejs Stack — pewnie z powodu zawłaszczenia tej nazwy w niesławnej implementacji java.util.Stack w Javie 1.0. Mimo dostępności java.util.Stack LinkedList umożliwia znacznie lepszą implementację stosu, dlatego będę zachęcał do stosowania net.mindviet.util.Stack.

Wybór preferowanej implementacji stosu można kontrolować również za pośrednictwem jawnej instrukcji importu:

```
import net.mindview.util.Stack;
```

Za sprawą takiego wiersza wszelkie odwołania do Stack będą dotyczyły wersji net.mindview.util.Stack, a pełnej kwalifikacji nazwy będzie z kolei wymagać klasa Stack z pakietu java.util.

Ćwiczenie 15. Stosy są często wykorzystywane do obliczania wyrażeń w językach programowania. Za pomocą klasy net.mindview.utils.Stack oblicz poniższe wyrażenie, w którym '+' oznacza „umieszczenie następnej litery na stosie", a '–' „zdjęcie szczytowego elementu stosu i wypisanie go na wyjściu" (4):

„+B+a+ł+–––+a+g+a–––+n–+w–+l+i+t–––+e–+r+k––+a+c+h–––".

Interfejs Set

Zbiór (ang. *set*) z definicji nie może zawierać więcej niż jednego egzemplarza danej wartości. Próba dodania kolejnych egzemplarzy identycznych obiektów zostanie zignorowana, co zapobiega dublowaniu elementów zbioru. Najpopularniejszym zastosowaniem kontenerów Set są testy przynależności do zbioru pozwalające na stwierdzenie obecności obiektu w zbiorze. Z tego powodu najważniejszą bodaj operacją Set jest wyszukiwanie elementów i ta właśnie operacja została zoptymalizowana pod kątem szybkości.

Set ma dokładnie ten sam interfejs co Collection, więc nie posiada żadnych dodatkowych funkcji, jak to było w przypadku dwóch różnych odmian List. Mimo że zbiór Set jest rozszerzeniem interfejsu Collection, zachowuje się inaczej (jest to modelowy przykład pogodzenia dziedziczenia i polimorfizmu — wyrażenie różnego zachowania). Set określa przynależność obiektu do zbioru na podstawie „wartości" obiektu, którą zajmiemy się bardziej szczegółowo przy okazji rozdziału „Kontenery z bliska".

Oto przykład użycia kontenera HashSet z obiektami Integer:

```
//: holding/SetOfInteger.java
import java.util.*;

public class SetOfInteger {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Set<Integer> intset = new HashSet<Integer>();
    for(int i = 0; i < 10000; i++)
      intset.add(rand.nextInt(30));
    System.out.println(intset);
  }
} /* Output:
[15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 29, 14, 24, 4, 19, 26, 11, 18, 3, 12, 27, 17, 2, 13, 28, 20, 25, 10, 5, 0]
*///:~
```

Do kontenera dodaliśmy dziesięć tysięcy wartości losowych z przedziału od 0 do 29, co pozwala stwierdzić na pewno, że wszystkie wartości zostały niejednokrotnie zdublowane. Mimo to zbiór zawiera jedynie po jednym egzemplarzu każdej wartości.

Jak widać, przy wypisywaniu zawartości zbioru nie została zachowana jakaś konkretna kolejność elementów. Otóż implementacja HashSet optymalizuje wydajność dostępu za pomocą techniki *haszowania* omówionej w rozdziale „Kontenery z bliska". Kolejność elementów w HashSet różni się od kolejności zachowywanej w TreeSet czy LinkedHashSet, bo wszystkie te implementacje opierają się na różnych strukturach przechowywania elementów. TreeSet utrzymuje wewnętrznie posortowaną strukturę drzewiastą, a HashSet rozkłada elementy wedle funkcji haszującej. LinkedHashSet również korzysta z takiej funkcji, ale na zewnątrz prezentuje porządek zgodny z kolejnością wstawiania, udając najzwyklejszą listę.

Jeśli wypisywane elementy zbioru mają być posortowane, należy zamiast HashSet użyć kontenera TreeSet:

```
//: holding/SortedSetOfInteger.java
import java.util.*;

public class SortedSetOfInteger {
  public static void main(String[] args) {
    Random rand = new Random(47);
    SortedSet<Integer> intset = new TreeSet<Integer>();
    for(int i = 0; i < 10000; i++)
      intset.add(rand.nextInt(30));
    System.out.println(intset);
  }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
*///:~
```

Jako się rzekło, jedną z częściej wykonywanych operacji na zbiorze jest sprawdzanie przynależności obiektu do zbioru realizowane metodą `contains()`; dostępne są jednak również operacje, które przypomną Ci diagramy Venna, rysowane jeszcze w szkole podstawowej:

```
//: holding/SetOperations.java
import java.util.*;
import static net.mindview.util.Print.*;

public class SetOperations {
  public static void main(String[] args) {
    Set<String> set1 = new HashSet<String>();
    Collections.addAll(set1,
      "A B C D E F G H I J K L".split(" "));
    set1.add("M");
    print("H: " + set1.contains("H"));
    print("N: " + set1.contains("N"));
    Set<String> set2 = new HashSet<String>();
    Collections.addAll(set2, "H I J K L".split(" "));
    print("set2 in set1: " + set1.containsAll(set2));
    set1.remove("H");
    print("set1: " + set1);
    print("set2 w set1: " + set1.containsAll(set2));
    set1.removeAll(set2);
    print("set2 usunięty z set1: " + set1);
    Collections.addAll(set1, "X Y Z".split(" "));
    print("'X Y Z' dodane do set1: " + set1);
  }
} /* Output:
H: true
N: false
set2 w set1: true
set1: [D, K, C, B, L, G, I, M, A, F, J, E]
set2 w set1: false
set2 usunięty z set1: [D, C, B, G, M, A, F, E]
'X Y Z' dodane do set1: [Z, D, C, B, G, M, A, F, Y, X, E]
*///:~
```

Nazwy metod mówią same za siebie; jeszcze kilka znajdziesz w dokumentacji JDK.

Zbiory mogą okazać się przydatne do generowania list wartości unikatowych. Załóżmy, że chcemy sporządzić wykaz wszystkich słów z pliku *SetOperations.java*. Plik ten wczytamy do kontenera Set za pomocą klasy `net.mindview.util.TextFile` prezentowanej w dalszej części książki:

```
//: holding/UniqueWords.java
import java.util.*;
import net.mindview.util.*;

public class UniqueWords {
  public static void main(String[] args) {
    Set<String> words = new TreeSet<String>(
      new TextFile("SetOperations.java", "\\W+"));
    System.out.println(words);
  }
} /* Output:
```

```
[A, B, C, Collections, D, E, F, G, H, HashSet, I, J, K, L, M, N, Output, Print, Set, SetOperations, String, X,
Y, Z, add, addAll, dodane, args, class, contains, containsAll, false, z, holding, import, w, java, main,
mindview, net, new, print, public, remove, removeAll, usuni, ty, set1, set2, split, static, do, true, util, void]
*///:~
```

Klasa `TextFile` dziedziczy po `List<String>`. Konstruktor `TextFile` otwiera plik i dzieli jego zawartość na słowa rozpoznawane na podstawie *wyrażenia regularnego* „\\W+", które oznacza „jedna albo wiele liter" (wyrażenia regularne omówimy w rozdziale „Ciągi znaków"). Wynik jest przekazywany do konstruktora `TreeSet`, który dodaje do własnej kolekcji zawartość otrzymanej listy. Ponieważ używamy `TreeSet`, elementy są sortowane według wartości. W tym przypadku oznacza to sortowanie *leksykograficzne*, z różnicowaniem wielkości liter. Gdyby chodziło o sortowanie *alfabetyczne* (bez różnicowania wielkości liter), należałoby do konstruktora `TreeSet` przekazać również obiekt komparatora `String.CASE_INSENSITIVE_ORDER` (*komparator* to obiekt ustalający wzajemny porządek elementów):

```
//: holding/UniqueWordsAlphabetic.java
// Producing an alphabetic listing.
import java.util.*;
import net.mindview.util.*;

public class UniqueWordsAlphabetic {
  public static void main(String[] args) {
    Set<String> words =
      new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
    words.addAll(
      new TextFile("SetOperations.java", "\\W+"));
    System.out.println(words);
  }
} /* Output:
[A, add, addAll, args, B, C, class, Collections, contains, containsAll, D, do, dodane, E, F, false, G, H,
HashSet, holding, I, import, J, java, K, L, M, main, mindview, N, net, new, Output, Print, public, remove,
removeAll, Set, set1, set2, SetOperations, split, static, String, true, ty, util, usuni, w, void, X, Y, z, Z]
*///:~
```

Komparatorom przyjrzymy się w rozdziale „Tablice".

Ćwiczenie 16. Utwórz zbiór samogłosek. Na bazie pliku *UniqueWords.java* zlicz i wypisz liczbę samogłosek w każdym słowie podawanym na wejście; wypisz też łączną liczbę samogłosek w pliku wejściowym (5).

Interfejs Map

Zdolność do odwzorowywania obiektów na inne obiekty to niezwykle istotna pomoc w rozwiązywaniu zadań programistycznych. Weźmy za przykład program, który ma badać losowość wyników generowanych przez klasę `Random` z biblioteki standardowej języka Java. `Random` powinno generować sekwencje pseudolosowe o równomiernym rozkładzie, ale aby to sprawdzić, należy wygenerować mnóstwo wartości oraz zliczyć wystąpienia poszczególnych wartości i sklasyfikować w pożądanych przedziałach. Wszystko to można załatwić kontenerem `Map`, który przechowywałby pary, gdzie kluczem byłaby wartość pseudolosowa generowana przez `Random`, a wartością — liczba wystąpień wartości pseudoolosowej:

```
// : holding/Statistics.java
// Prosta demonstracja kontenera HashMap.
import java.util.*;

public class Statistics {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Map<Integer,Integer> m =
      new HashMap<Integer,Integer>();
    for(int i = 0; i < 10000; i++) {
      // Losowanie liczby z zakresu od 0 do 20:
      int r = rand.nextInt(20);
      Integer freq = m.get(r);
      m.put(r, freq == null ? 1 : freq + 1);
    }
    System.out.println(m);
  }
} /* Output:
{15=497, 4=481, 19=464, 8=468, 11=531, 16=533, 18=478, 3=508, 7=471, 12=521, 17=509, 2=489,
13=506, 9=549, 6=519, 1=502, 14=477, 10=513, 5=503, 0=481}
*///:~
```

W metodzie `main()` dochodzi do automatycznej konwersji wygenerowanej wartości int na postać referencji do klasy `Integer`, które można wstawiać do `HashMap` (kontenery nie mogą przechowywać wartości typów podstawowych). Metoda `get()` dla kluczy nieobecnych jeszcze w odwzorowaniu (co oznacza, że para o takim kluczu jest wstawiana pierwszy raz) zwraca wartość `null`; dla pozostałych kluczy `get()` zwraca skojarzoną z kluczem wartość Integer, która jest od razu inkrementowana (znów przy wydatnej pomocy ze strony mechanizmu automatycznego pakowania wartości podstawowych w obiekty).

Oto przykład, w którym możemy wykorzystać opis (`String`) do wyszukania obiektu zwierzaka (`Pet`). Program pokazuje też, że kontener `Map` można przeszukiwać w poszukiwaniu klucza albo wartości za pomocą metod `containsKey()` i `containsValue()`:

```
// : holding/PetMap.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class PetMap {
  public static void main(String[] args) {
    Map<String,Pet> petMap = new HashMap<String,Pet>();
    petMap.put("Mój kot", new Cat("Molly"));
    petMap.put("Mój pies", new Dog("Ginger"));
    petMap.put("Mój chomik", new Hamster("Bosco"));
    print(petMap);
    Pet dog = petMap.get("Mój pies");
    print(dog);
    print(petMap.containsKey("Mój pies"));
    print(petMap.containsValue(dog));
  }
} /* Output:
{Mój kot=Cat Molly, Mój chomik=Hamster Bosco, Mój pies=Dog Ginger}
Dog Ginger
true
true
*///:~
```

Kontenery `Map`, tak jak tablice i kolekcje `Collection`, mogą być łatwo rozbudowywane do wielu wymiarów; wystarczy utworzyć kontener `Map`, którego elementami są inne kontenery `Map` (te z kolei mogą przechowywać elementy będące jeszcze innymi kontenerami, również typu `Map`, i tak dalej). Jak widać, w prosty sposób można zmontować kontenery w całkiem pokaźne struktury danych. Załóżmy na przykład, że chcemy rejestrować osoby posiadające wiele zwierząt — wystarczy nam do tego kontener `Map<Person, List<Pet>>`:

```
//: holding/MapOfList.java
package holding;
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class MapOfList {
  public static Map<Person, List<? extends Pet>>
    petPeople = new HashMap<Person, List<? extends Pet>>();
  static {
    petPeople.put(new Person("Tomasz"),
      Arrays.asList(new Cymric("Molly"),new Mutt("Spot")));
    petPeople.put(new Person("Kasia"),
      Arrays.asList(new Cat("Szoruś"),
        new Cat("Mała Lu"), new Dog("Marten")));
    petPeople.put(new Person("Marysia"),
      Arrays.asList(
       new Pug("Lolo vel Leonard Moppsen"),
       new Cat("Stefan vel Czarny Łobuz"),
       new Cat("Pinkola")));
    petPeople.put(new Person("Lucek"),
      Arrays.asList(new Rat("Gonek"), new Rat("Lelek")));
    petPeople.put(new Person("Jakub"),
      Arrays.asList(new Rat("Kolka")));
  }
  public static void main(String[] args) {
    print("Osoby: " + petPeople.keySet());
    print("Zwierzaki: " + petPeople.values());
    for(Person person : petPeople.keySet()) {
      print(person + " ma:");
      for(Pet pet : petPeople.get(person))
        print("    " + pet);
    }
  }
} /* Output:
Osoby: [Person Lucek, Person Marysia, Person Jakub, Person Tomasz, Person Kasia]
Zwierzaki: [[Rat Gonek, Rat Lelek], [Pug Lolo vel Leonard Moppsen, Cat Stefan vel Czarny Łobuz,
Ca  Pinkola], [Rat Kolka], [Cymric Molly, Mutt Spot], [Cat Szoruś, Cat Mała Lu, Dog Marten]]
Person Lucek ma:
   Rat Gonek
   Rat Lelek
Person Marysia ma:
   Pug Lolo vel Leonard Moppsen
   Cat Stefan vel Czarny Łobuz
   Cat Pinkola
Person Jakub has:
   Rat Kolka
Person Tomasz has:
   Cymric Molly
   Mutt Spot
```

```
Person Kasia has:
  Cat Szoruś
  Cat Mała Lu
  Dog Marten
*///:~
```

Kontener Map może zwracać zbiór (Set) kluczy, kolekcję (Collection) wartości albo zbiór par. Metoda keySet() generuje zbiór Set z kompletem kluczy petPeople; zwrócony zbiór jest wykorzystywany w pętli foreach do przeglądania zawartości odwzorowania.

Ćwiczenie 17. Weź klasę Gerbil z ćwiczenia 1. i umieść jej obiekty w kontenerze Map, kojarząc każdy egzemplarz Gerbil (wartość) z nazwą („Gonek" czy „Lelek" itd.) w postaci obiektu String (klucz). Pozyskaj iterator zbioru zwracanego przez keySet() i wykorzystaj go do przejrzenia kontenera Map; wyłuskaj z kontenera obiekty Gerbil odpowiadające wszystkim kluczom i wypisz je na wyjściu, oraz wywołaj dla każdego z nich metodę hop() (2).

Ćwiczenie 18. Wypełnij kontener HashMap parami klucz-wartość. Wypisz wyniki, ujawniając efekty porządkowania na podstawie funkcji haszującej. Wyodrębnij z kontenera pary, posortuj je według kluczy i umieść całość w kontenerze LinkedHashMap. Pokaż, że tym razem zachowana została pierwotna kolejność elementów (3).

Ćwiczenie 19. Powtórz poprzednie ćwiczenie z klasami HashSet i LinkedHashSet (2).

Ćwiczenie 20. Zmień ćwiczenie 16. tak, aby rejestrować liczbę wystąpień każdej samogłoski (3).

Ćwiczenie 21. Za pomocą kontenera Map<String, Integer> i naśladując kod z pliku *UniqueWords.java*, napisz program, który zliczy wystąpienia słów w pliku. Posortuj wyniki za pomocą metody Collections.sort() z argumentem String.CASE_INSENSITIVE_ORDER (wymuszając sortowanie alfabetyczne) i wypisz wyniki na wyjście programu (3).

Ćwiczenie 22. Zmień poprzednie ćwiczenie tak, aby wykorzystywało klasę zawierającą pole typu String i pole licznika wystąpień słowa; lista słów ma mieć postać obiektów tej klasy (dla każdego słowa z osobna) umieszczanych w zbiorze Set (5).

Ćwiczenie 23. Na bazie programu *Statistics.java* napisz program, który powtórzy test wielokrotnie i sprawdzi, czy któreś z liczb pojawiają się częściej niż inne (4).

Ćwiczenie 24. Wypełnij kontener LinkedHashMap kluczami String i wartościami wybranego typu. Wyłuskaj z kontenera poszczególne pary, uporządkuj je według wartości kluczy i ponownie wstaw do kontenera Map (2).

Ćwiczenie 25. Utwórz kontener Map<String, ArrayList<Integer>>. Użyj klasy net.mindview.util.TextFile do otwarcia pliku i wczytania z niego poszczególnych słów (użyj wyrażenia regularnego „\\W+" jako drugiego argumentu wywołania konstruktora TextFile). Wczytując słowa, zliczaj je i dla każdego słowa z pliku zarejestruj w ArrayList<Integer> licznik słów przypadający na to słowo — będzie on reprezentował pozycję słowa w pliku (3).

Ćwiczenie 26. Weź wynikowy kontener z poprzedniego ćwiczenia i odtwórz pierwotną kolejność słów w pliku (4).

Interfejs Queue

Kolejka (ang. *queue*) to kontener typu *first-in, first-out* (FIFO). Znaczy to, że elementy dodaje się na koniec, a pobiera z początku, a kolejność wkładania i wyjmowania elementów jest taka sama. Kolejki są zwykle wykorzystywane w roli pewnego mechanizmu transferu obiektów pomiędzy różnymi obszarami programu. Są one szczególnie przydatne w programowaniu współbieżnym, bowiem pozwalają bezpiecznie przekazywać obiekty pomiędzy zadaniami — przekonasz się o tym w rozdziale „Współbieżność".

`LinkedList` zawiera metody odpowiadające zachowaniu kolejki i implementuje interfejs `Queue`, więc `LinkedList` można użyć w roli implementacji kolejki. Rzutowanie w górę obiektu `LinkedList` na typ `Queue` w poniższym przykładzie pozwala na zaprezentowanie metod charakterystycznych właśnie dla interfejsu `Queue`:

```
//: holding/QueueDemo.java
// Rzutowanie w górę LinkedList na typ Queue.
import java.util.*;

public class QueueDemo {
  public static void printQ(Queue queue) {
    while(queue.peek() != null)
      System.out.print(queue.remove() + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    Queue<Integer> queue = new LinkedList<Integer>();
    Random rand = new Random(47);
    for(int i = 0; i < 10; i++)
      queue.offer(rand.nextInt(i + 10));
    printQ(queue);
    Queue<Character> qc = new LinkedList<Character>();
    for(char c : "Brontozaur".toCharArray())
      qc.offer(c);
    printQ(qc);
  }
} /* Output:
8 1 1 1 5 14 3 1 0 1
B r o n t o z a u r
*///:~
```

Jedną z metod charakterystycznych dla kolejki jest `offer()`. Metoda ta wstawia element na koniec kolejki, o ile jest to możliwe — w przeciwnym przypadku zwraca wartość false. Metody `peek()` i `element()` zwracają element z przodu kolejki *bez jego usuwania z kolejki*, tyle że jeśli kolejka jest pusta, `peek()` zwraca wartość false, a `element()` zgłasza wyjątek `NoSuchElementException`. Z kolei metody `poll()` i `remove()` zwracają usunięty element z czoła kolejki, również różniąc się jedynie zachowaniem przy braku elementów: `poll()` zwraca wtedy false, a `remove()` zgłasza wyjątek `NoSuchElementException`.

Mechanizm automatycznego pakowania wartości podstawowych umieszcza wartości int uzyskiwane za pomocą metody `nextInt()` w obiektach typu `Integer`, nadających się do umieszczenia w kolejce queue; wartości znakowe (`char`) są z kolei konwertowane na typ

Character, wymagany przez qc. Interfejs Queue zawęża zestaw metod LinkedList do metod właściwych dla kolejek, nie można więc korzystać z wywołań metod LinkedList (chyba że po uprzednim rzutowaniu Queue z powrotem na LinkedList).

Zauważ, że metody charakterystyczne dla interfejsu Queue stanowią kompletny i samodzielny zestaw — dysponujemy w pełni funkcjonalną kolejką bez uciekania się do metod z interfejsu Collection.

Ćwiczenie 27. Napisz klasę o nazwie Command, która zawiera ciąg znaków String i metodę operation(), która go wypisuje. Napisz drugą klasę, z metodą wypełniającą kolejkę Queue obiektami klasy Command i zwracającą wypełniony kontener. Przekaż kontener do metody z trzeciej klasy; metoda ta ma skonsumować obiekty z kolejki Queue, wywołując dla każdego z nich metodę operation() (2).

PriorityQueue

Najbardziej typową *dyscypliną kolejkowania* jest FIFO. Dyscyplina kolejkowania decyduje o kolejności wydobywania elementów z kolejki. FIFO (ang. *first-in, first-out*) oznacza, że następnym elementem będzie ten, który najdłużej przebywa w kolejce (czyli „kto pierwszy, ten lepszy").

Kolejka priorytetowa (ang. *priority queue*) przewiduje, że następnym elementem wydobytym z kolejki będzie element o najwyższym priorytecie. Na przykład na lotnisku można by wyciągnąć z kolejki tego oczekującego, którego samolot ma za chwilę odlecieć. Dalej, w systemach opartych na wymianie komunikatów niektóre komunikaty są ważniejsze od innych i powinny być obsłużone jak najwcześniej, niezależnie od czasu przybycia. Java SE5 oferuje klasę PriorityQueue, która automatycznie implementuje taką właśnie dyscyplinę kolejkowania.

Kiedy za pomocą metody offer() umieszczamy obiekt w kolejce PriorityQueue, obiekt jest wstawiany na miejsce zgodne z jego priorytetem[5]. Domyślnie obiekty są układane według tak zwanego *porządku naturalnego*, który można zmienić, udostępniając własny komparator (obiekt klasy implementującej interfejs Comparator). Kolejka priorytetowa zapewnia, że wywołanie peek(), pool() bądź remove() zwróci element o najwyższym priorytecie.

Przystosowanie kolejki priorytetowej do obsługi wartości typów wbudowanych, jak Integer, String i Character, jest banalne. W poniższym przykładzie w roli pierwszej serii elementów wykorzystamy wartości losowe identyczne z tymi z poprzedniego przykładu — można więc będzie zobaczyć, że są porządkowane inaczej niż w zwykłej kolejce FIFO:

```
//: holding/PriorityQueueDemo.java
import java.util.*;
```

[5] Choć akurat ten aspekt zachowania PriorityQueue jest faktycznie zależny od implementacji. Algorytmy kolejek priorytetowych zazwyczaj porządkują elementy kolejki przy ich wstawianiu, ale selekcja elementu o najwyższym priorytecie może równie dobrze odbywać się dopiero przy wyjmowaniu elementu. Wybór algorytmu może być istotny w przypadku, kiedy obiekty wstawiane do kolejki mogą w czasie oczekiwania na wyjęcie zmieniać priorytet.

```
public class PriorityQueueDemo {
  public static void main(String[] args) {
    PriorityQueue<Integer> priorityQueue =
      new PriorityQueue<Integer>();
    Random rand = new Random(47);
    for(int i = 0; i < 10; i++)
      priorityQueue.offer(rand.nextInt(i + 10));
    QueueDemo.printQ(priorityQueue);

    List<Integer> ints = Arrays.asList(25, 22, 20,
      18, 14, 9, 3, 1, 1, 2, 3, 9, 14, 18, 21, 23, 25);
    priorityQueue = new PriorityQueue<Integer>(ints);
    QueueDemo.printQ(priorityQueue);
    priorityQueue = new PriorityQueue<Integer>(
        ints.size(), Collections.reverseOrder());
    priorityQueue.addAll(ints);
    QueueDemo.printQ(priorityQueue);

    String fact = "BEZ PRACY NIE MA KOŁACZY";
    List<String> strings = Arrays.asList(fact.split(""));
    PriorityQueue<String> stringPQ =
      new PriorityQueue<String>(strings);
    QueueDemo.printQ(stringPQ);
    stringPQ = new PriorityQueue<String>(
      strings.size(), Collections.reverseOrder());
    stringPQ.addAll(strings);
    QueueDemo.printQ(stringPQ);

    Set<Character> charSet = new HashSet<Character>();
    for(char c : fact.toCharArray())
      charSet.add(c); // Automatyczne pakowanie w obiekty
    PriorityQueue<Character> characterPQ =
      new PriorityQueue<Character>(charSet);
    QueueDemo.printQ(characterPQ);
  }
} /* Output:
0 1 1 1 1 1 3 5 8 14
1 1 2 3 3 9 9 14 14 18 18 20 21 22 23 25 25
25 25 23 22 21 20 18 18 14 14 9 9 3 3 2 1 1
    A A A B C C E E I K M N O P R Y Y Z Z Ł
Ł Z Z Y Y R P O N M K I E E C C B A A A
 A B C E I K M N O P R Y Z Ł
  *///:~
```

Jak widać, kolejka może przechowywać elementy zdublowane, a najwyższy priorytet mają wartości najniższe (w przypadku elementów typu `String` spacje również liczą się jako wartości i dlatego mają większy priorytet niż znaki). Aby sprawdzić, jak zmienia się kolejność elementów po wskazaniu własnego komparatora, w trzecim wywołaniu konstruktora `PriorityQueue<Integer>` i drugim wywołaniu `PriorityQueue<String>` przekazaliśmy komparator odwracający porządek wygenerowany przez metodę `Collections.reverseOrder()` (nowość w Javie SE5).

Ostatni fragment programu demonstruje zastosowanie zbioru `HashSet` do wyeliminowania duplikatów znaków.

Obiekty klas `Integer`, `String` i `Character` nadają się do umieszczania w kolejkach `PriorityQueue`, bo te klasy mają wbudowane mechanizmy porządkowania. Ale aby wykorzystać w kolejce priorytetowej obiekty własnych klas, trzeba albo uzupełnić te klasy o funkcje określające wzajemny porządek egzemplarzy klasy, albo skonstruować kolejkę priorytetową z obiektem komparatora. Zobaczymy to na nieco bardziej rozbudowanym przykładzie w rozdziale „Kontenery z bliska".

Ćwiczenie 28. Wypełnij kolejkę priorytetową (za pomocą metody `offer()`) wartościami typu `Double` generowanymi przez stosowną metodę klasy `java.util.Random`; wyciągnij kolejne elementy z kolejki za pomocą metody `poll()` i wypisz je na wyjściu programu (2).

Ćwiczenie 29. Utwórz prostą klasę, która dziedziczy po klasie `Object` i nie posiada żadnych pól składowych; wykaż, że nie można skutecznie dodać kilku egzemplarzy takiej klasy do kolejki priorytetowej `PriorityQueue`. Wyjaśnienie tego fenomenu znajdziesz w rozdziale „Kontenery z bliska" (2).

Collection kontra Iterator

`Collection` to podstawowy interfejs opisujący wspólne cechy wszystkich kontenerów sekwencyjnych. Można by go traktować jako „interfejs wypadkowy", wynikający z pokrywania się innych interfejsów. Do tego domyślną implementację interfejsu `Collection` udostępnia klasa `java.util.AbstractCollection`; możemy więc tworzyć nowe własne podtypy `AbstractCollection` bez niepotrzebnego powielania kodu.

Jednym z uzasadnień dla posiadania interfejsu jest zwiększenie stopnia ogólności kodu. Komunikując się z interfejsem, a nie z implementacją, możemy stosować ten sam kod do obiektów różnych typów[6]. Jeśli więc napiszę metodę wymagającą przekazania `Collection`, metoda ta będzie mogła obsługiwać wszelkie typy implementujące interfejs `Collection` — a to pozwoli na wybór implementacji `Collection` w nowej klasie pod kątem wykorzystania w tejże metodzie. Warto tutaj zaznaczyć, że standardowa biblioteka języka C++ nie wyodrębnia wspólnej klasy bazowej kontenerów — wspólnota zachowania kontenerów jest wyrażana zbiorem iteratorów. W języku Java można by naśladować podejście zastosowanie w C++, a więc wyrażanie wspólnoty kontenerów właśnie iteratorami, a nie wspólnym interfejsem `Collection`, ale oba podejścia są ze sobą powiązane, bo wszak implementacja `Collection` oznacza równoczesne zdefiniowanie metody `iterator()`:

```
//: holding/InterfaceVsIterator.java
import typeinfo.pets.*;
import java.util.*;

public class InterfaceVsIterator {
  public static void display(Iterator<Pet> it) {
    while(it.hasNext()) {
      Pet p = it.next();
```

[6] Niektórzy postulują automatyczne tworzenie interfejsu dla każdej możliwej kombinacji metod w klasie — nawet dla każdej pojedynczej klasy. Osobiście uważam, że interfejs powinien coś znaczyć, a nie tylko mechanicznie powielać kombinacje metod, więc z wyodrębnieniem interfejsu wstrzymuję się do momentu, w którym widzę potrzebę i korzyści z jego obecności.

```
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
  }
  public static void display(Collection<Pet> pets) {
    for(Pet p : pets)
      System.out.print(p.id() + ":" + p + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    List<Pet> petList = Pets.arrayList(8);
    Set<Pet> petSet = new HashSet<Pet>(petList);
    Map<String,Pet> petMap =
      new LinkedHashMap<String,Pet>();
    String[] names = ("Rychu, Eryk, Robin, Lucek, " +
      "Basia, Lubomir, Spot, Łapek").split(", ");
    for(int i = 0; i < names.length; i++)
      petMap.put(names[i], petList.get(i));
    display(petList);
    display(petSet);
    display(petList.iterator());
    display(petSet.iterator());
    System.out.println(petMap);
    System.out.println(petMap.keySet());
    display(petMap.values());
    display(petMap.values().iterator());
  }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
{Rychu=Rat, Eryk=Manx, Robin=Cymric, Lucek=Mutt, Basia=Pug, Lubomir=Cymric, Spot=Pug,
Łapek=Manx}
[Rychu, Eryk, Robin, Lucek, Basia, Lubomir, Spot, Łapek]
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:~
```

Obie wersje metody `display()` przyjmują obiekty typu `Map`, jak i podtypów `Collection`, przy czym interfejsy `Collection` i `Iterator` zwalniają metody `display()` ze znajomości szczegółów implementacji konkretnego kontenera, na którym operują.

W tym przypadku oba podejścia dają ten sam efekt. W rzeczy samej `Collection` jest nawet nieco lepszy, bo to typ kategorii `Iterable`, więc w implementacji wersji `display()` dla interfejsu Collection można stosować pętlę foreach, co nieco zwiększa czytelność kodu.

Zastosowanie interfejsu `Iterator` staje się wyzwaniem przy implementacji klasy obcej, która nie implementuje `Collection` i w której implementacja `Collection` byłaby albo trudna, albo po prostu uciążliwa. Gdybyśmy na przykład utworzyli implementację `Collection` przez dziedziczenie po klasie przechowującej obiekty `Pet`, musielibyśmy zaimplementować w niej wszystkie metody `Collection`, mimo że w metodzie `display()` w ogóle nie byłyby wykorzystywane. Co prawda implementacja mogłaby się sprowadzać do dziedziczenia po klasie `AbstractCollection`, ale nie uniknęlibyśmy definiowania metody `iterator()` oraz `size()` jako metod nieimplementowanych w `AbstractCollection`, a wykorzystywanych przez inne metody `AbstractCollection`:

```
//: holding/CollectionSequence.java
import typeinfo.pets.*;
import java.util.*;

public class CollectionSequence
extends AbstractCollection<Pet> {
  private Pet[] pets = Pets.createArray(8);
  public int size() { return pets.length; }
  public Iterator<Pet> iterator() {
    return new Iterator<Pet>() {
      private int index = 0;
      public boolean hasNext() {
        return index < pets.length;
      }
      public Pet next() { return pets[index++]; }
      public void remove() { // Niezaimplementowana
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    CollectionSequence c = new CollectionSequence();
    InterfaceVsIterator.display(c);
    InterfaceVsIterator.display(c.iterator());
  }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:~
```

Metoda `remove()` to przypadek „operacji opcjonalnej", o których dowiemy się więcej w rozdziale „Kontenery z bliska". Nie ma tu potrzeby jej implementowania, a w razie wywołania spowoduje ona zgłoszenie wyjątku.

Przykład pokazuje, że implementacja `Collection` oznacza również implementację metody `iterator()`, a samo definiowanie tej metody wymaga tylko nieco mniej wysiłku programistycznego niż dziedziczenie po klasie `AbstractCollection`. Ale jeśli dana klasa już dziedziczy po innej klasie, to nie może równocześnie dziedziczyć po `AbstractCollection`. Wtedy implementacja `Collection` oznaczałaby konieczność zdefiniowania kompletu metod tego interfejsu. W takim przypadku znacznie łatwiej byłoby jednak ograniczyć się do dziedziczenia i zadbania o możliwość utworzenia iteratora:

```
//: holding/NonCollectionSequence.java
import typeinfo.pets.*;
import java.util.*;

class PetSequence {
  protected Pet[] pets = Pets.createArray(8);
}

public class NonCollectionSequence extends PetSequence {
  public Iterator<Pet> iterator() {
    return new Iterator<Pet>() {
      private int index = 0;
      public boolean hasNext() {
        return index < pets.length;
```

```
      }
      public Pet next() { return pets[index++]; }
      public void remove() { // Niezaimplementowana
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    NonCollectionSequence nc = new NonCollectionSequence();
    InterfaceVsIterator.display(nc.iterator());
  }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:~
```

Wygenerowanie iteratora jest najmniej wiążącą metodą połączenia sekwencji z metodą przetwarzającą sekwencję, narzuca też znacznie mniej więzów na klasy sekwencji niż implementowanie interfejsu `Collection`.

Ćwiczenie 30. Zmień plik *CollectionSequence.java* tak, aby klasa nie dziedziczyła po `AbstractCollection`, ale implementowała interfejs `Collection` (5).

Iteratory a pętle foreach

Jak dotąd pętli foreach używaliśmy głównie z tablicami, ale da się je stosować również z dowolnymi implementacjami `Collection`. Widziałeś już kilka przykładów z użyciem `ArrayList`; oto ogólniejsze potwierdzenie powyższego stwierdzenia:

```
//: holding/ForEachCollections.java
// Wszystkie kolekcje można stosować z foreach.
import java.util.*;

public class ForEachCollections {
  public static void main(String[] args) {
    Collection<String> cs = new LinkedList<String>();
    Collections.addAll(cs,
      "Nie ma jak w domu".split(" "));
    for(String s : cs)
      System.out.print("'" + s + "' ");
  }
} /* Output:
'Nie' 'ma' 'jak' 'w' 'domu'
*///:~
```

Skoro `cs` to obiekt typu `Collection`, powyższy kod dowodzi przydatności składni foreach do przeglądania dowolnych kolekcji.

Całość działa, ponieważ Java SE5 zawiera nowy interfejs o nazwie `Iterable`, który udostępnia metodę `iterator()` generującą `Iterator`; pętla foreach wykorzystuje do przeglądania sekwencji właśnie interfejs `Iterable`. Jeśli więc utworzysz dowolną klasę implementującą `Iterable`, będziesz mógł zastosować do jej obiektów pętlę foreach:

```
// : holding/IterableClass.java
// Z foreach działa wszystko, co implementuje interfejs Iterable.
import java.util.*;

public class IterableClass implements Iterable<String> {
  protected String[] words = ("I stąd właśnie wiemy, że " +
    "Ziemia ma kształt banana.").split(" ");
  public Iterator<String> iterator() {
    return new Iterator<String>() {
      private int index = 0;
      public boolean hasNext() {
        return index < words.length;
      }
      public String next() { return words[index++]; }
      public void remove() { // Niezaimplementowana
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    for(String s : new IterableClass())
      System.out.print(s + " ");
  }
} /* Output:
I stąd właśnie wiemy, że Ziemia ma kształt banana.
*///:~
```

Metoda `iterator()` zwraca egzemplarz anonimowej wewnętrznej implementacji `Iterator<String>` dostarczającej pojedyncze elementy tablicy. W metodzie `main()` można zaobserwować, że klasa `IterableClass` faktycznie może występować w roli sekwencji źródłowej pętli foreach.

W Javie SE5 interfejs `Iterable` jest implementowany przez cały szereg klas, w tym wszystkie klasy `Collection` (ale nie klasy odwzorowań `Map`). Spójrzmy na przykładowy program wypisujący komplet zmiennych środowiskowych systemu operacyjnego:

```
// : holding/EnvironmentVariables.java
import java.util.*;

public class EnvironmentVariables {
  public static void main(String[] args) {
    for(Map.Entry entry: System.getenv().entrySet()) {
      System.out.println(entry.getKey() + ": " +
        entry.getValue());
    }
  }
} /* (Execute to see output) *///:~
```

Metoda `System.getenv()`[7] zwraca kontener `Map`, `entrySet()` generuje zbiór `Set` elementów `Map.Entry`, a zbiór `Set` implementuje `Iterable` i jako taki może występować w pętlach foreach.

[7] Metoda ta nie była dostępna przed wydaniem Javy SE5, bo uważano, że wprowadza zbyt ścisłe powiązanie z systemem operacyjnym, a więc w jakiś sposób jest sprzeczna z regułą maksymalnej przenośności kodu Javy. Jej włączenie do nowego wydania świadczy o przyjęciu przez projektantów języka postawy mniej ideologicznej, a bardziej pragmatycznej.

Składnia foreach działa z tablicami i wszelkimi klasami implementującymi `Iterable`, nie oznacza to jednak, że każda tablica jest automatycznie implementacją interfejsu `Iterable` ani że zachodzi automatyczne pakowanie wartości w obiekty:

```
//: holding/ArrayIsNotIterable.java
import java.util.*;

public class ArrayIsNotIterable {
  static <T> void test(Iterable<T> ib) {
    for(T t : ib)
      System.out.print(t + " ");
  }
  public static void main(String[] args) {
    test(Arrays.asList(1, 2, 3));
    String[] strings = { "A", "B", "C" };
    // Tablica działa w foreach, ale nie jest implementacją Iterable:
    //! test(strings);
    // Trzeba jawnie skonwertować ją na Iterable:
    test(Arrays.asList(strings));
  }
} /* Output:
1 2 3 A B C
*///:~
```

Próba przekazania tablicy jako `Iterable` zawiedzie, bowiem nie istnieje automatyczna konwersja tablic na typ `Iterable` — trzeba ją przeprowadzić ręcznie.

Ćwiczenie 31. Zmodyfikuj *polymorphism/shape/RandomShapeGenerator.java* tak, aby klasa generatora implementowała `Iterable`. Trzeba w tym celu dodać konstruktor przyjmujący liczbę elementów, które iterator ma wygenerować przed wyczerpaniem pętli. Sprawdź, jak działa nowa implementacja (3).

Idiom metody-adaptera

Co zrobić, jeśli do dyspozycji jest klasa implementująca interfejs `Iterable` i chcielibyśmy uzupełnić ją o dodatkowe sposoby używania klasy w pętli foreach? Na przykład, aby można było wybierać przeglądanie listy słów w przód albo wspak. Jeśli ograniczymy się do dziedziczenia po owej klasie i przesłonięcia metody `iterator()`, zastąpimy poprzednią wersję i nie uzyskamy możliwości wyboru.

Jednym z rozwiązań jest technika, którą określam mianem idiomu *metody-adaptera*. „Adapter" to zapożyczenie z wzorców projektowych, bo wymagania pętli foreach narzucają konkretny interfejs. Adapter rozwiązuje problem posiadania jednego interfejsu i potrzeby uzyskania innego. Chciałem dodać do klasy możliwość generowania iteratora wstecznego, nie tracąc możliwości pozyskiwania iteratora domyślnego — przesłanianie nie wchodziło więc w rachubę. Dlatego dodałem do klasy metodę, która generuje obiekt `Iterable` nadający się do stosowania w pętli foreach. Zaraz się okaże, że ta technika pozwala na rozmaite definiowanie iteracji w ramach pętli foreach:

```
//: holding/AdapterMethodIdiom.java
// Idiom "metoda-adapter" pozwala na stosowanie foreach
// z dodatkowymi rodzajami iteratorów.
import java.util.*;
```

```
class ReversibleArrayList<T> extends ArrayList<T> {
  public ReversibleArrayList(Collection<T> c) { super(c); }
  public Iterable<T> reversed() {
    return new Iterable<T>() {
      public Iterator<T> iterator() {
        return new Iterator<T>() {
          int current = size() - 1;
          public boolean hasNext() { return current > -1; }
          public T next() { return get(current--); }
          public void remove() { // Niezaimplementowana
            throw new UnsupportedOperationException();
          }
        };
      }
    };
  }
}

public class AdapterMethodIdiom {
  public static void main(String[] args) {
    ReversibleArrayList<String> ral =
      new ReversibleArrayList<String>(
        Arrays.asList("Być albo nie być".split(" ")));
    // Pozyskanie zwykłego iteratora wywołaniem iterator():
    for(String s : ral)
      System.out.print(s + " ");
    System.out.println();
    // A teraz iteratora alternatywnego
    for(String s : ral.reversed())
      System.out.print(s + " ");
  }
} /* Output:
Być albo nie być
być nie albo Być
*///:~
```

Jeśli w pętli foreach umieścimy po prostu obiekt ral, uzyskamy iterację domyślną, czyli w przód sekwencji. Ale jeśli na rzecz obiektu wywołamy metodę `reversed()`, zaobserwujemy iterację wspak sekwencji.

Idąc tym samym tropem, dodam dwie metody-adaptery do klasy z przykładu *IterableClass.java*:

```
//: holding/MultiIterableClass.java
// Kilka metod-adapterów.
import java.util.*;

public class MultiIterableClass extends IterableClass {
  public Iterable<String> reversed() {
    return new Iterable<String>() {
      public Iterator<String> iterator() {
        return new Iterator<String>() {
          int current = words.length - 1;
          public boolean hasNext() { return current > -1; }
          public String next() { return words[current--]; }
          public void remove() { // Niezaimplementowane
            throw new UnsupportedOperationException();
```

```
        }
      };
    }
  };
}
public Iterable<String> randomized() {
  return new Iterable<String>() {
    public Iterator<String> iterator() {
      List<String> shuffled =
        new ArrayList<String>(Arrays.asList(words));
      Collections.shuffle(shuffled, new Random(47));
      return shuffled.iterator();
    }
  };
}
public static void main(String[] args) {
  MultiIterableClass mic = new MultiIterableClass();
  for(String s : mic.reversed())
    System.out.print(s + " ");
  System.out.println();
  for(String s : mic.randomized())
    System.out.print(s + " ");
  System.out.println();
  for(String s : mic)
    System.out.print(s + " ");
}
} /* Output:
banana. kształt ma Ziemia że wiemy, właśnie stąd i
kształt i że banana. stąd Ziemia ma wiemy, właśnie
i stąd właśnie wiemy, że Ziemia ma kształt banana.
*///:~
```

Zauważ, że druga metoda-adapter (`randomized()`) nie tworzy iteratora, a jedynie zwraca iterator losowo potasowanej listy.

Na wyjściu programu widać, że metoda `Collections.shuffle()` nie ingeruje w ułożenie elementów oryginalnej tablicy, a jedynie przestawia miejscami referencje w zwracanej tablicy `shuffled`. Otóż metoda `randomized()` opakowuje wynik wywołania `Arrays.asList()` w kontenerze `ArrayList`. Gdyby kontener generowany przez `Arrays.asList()` był tasowany wprost, doszłoby do modyfikacji pierwotnej tablicy, jak tutaj:

```
//: holding/ModifyingArraysAsList.java
import java.util.*;

public class ModifyingArraysAsList {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Integer[] ia = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    List<Integer> list1 =
      new ArrayList<Integer>(Arrays.asList(ia));
    System.out.println("Przed tasowaniem: " + list1);
    Collections.shuffle(list1, rand);
    System.out.println("Po tasowaniu: " + list1);
    System.out.println("tablica: " + Arrays.toString(ia));

    List<Integer> list2 = Arrays.asList(ia);
    System.out.println("Przed tasowaniem: " + list2);
```

```
    Collections.shuffle(list2, rand);
    System.out.println("Po tasowaniu: " + list2);
    System.out.println("tablica: " + Arrays.toString(ia));
  }
} /* Output:
Przed tasowaniem: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Po tasowaniu: [4, 6, 3, 1, 8, 7, 2, 5, 10, 9]
tablica: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Przed tasowaniem: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Po tasowaniu: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
tablica: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
*///:~
```

W pierwszym przypadku wynik wywołania `Arrays.asList()` jest przekazywany do konstruktora klasy `ArrayList`, który tworzy obiekt `ArrayList` z referencjami elementów z tablicy `ia`. Tasowanie tych referencji nie modyfikuje oryginalnej tablicy. Ale jeśli wynik `Arrays.asList(ia)` zostanie użyty wprost, tasowanie zmieni kolejność elementów w `ia`. Ważne, aby pamiętać, że metoda `Arrays.asList()` generuje listę obiektów związaną z fizyczną implementacją tablicy, na której operowała metoda. Jakiekolwiek modyfikacje owej listy, jeśli nie mają wpływać na pierwotną tablicę, powinny być poprzedzone wykonaniem kopii listy w innym kontenerze.

Ćwiczenie 32. Wzorując się na przykładzie `MultiIterableClass`, dodaj do klasy z pliku *NonCollectionSequence.java* metody `reversed()` i `randomized()`, a także uczyń klasę `NonCollectionSequence` implementacją interfejsu Iterable; wykaż, że wszystkie te dodatki działają w pętlach foreach (2).

Podsumowanie

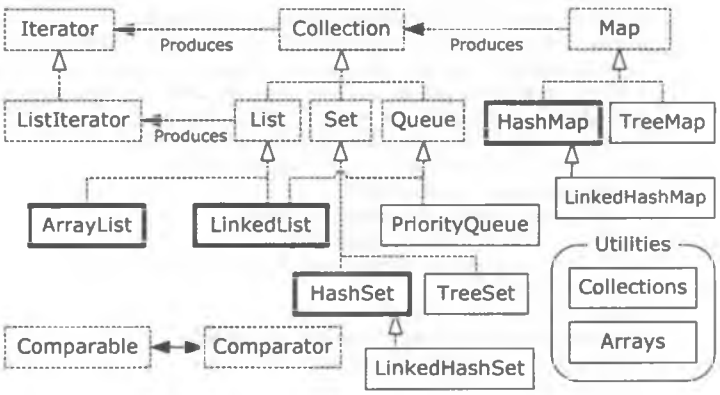
Podsumujmy wiadomości o przechowywaniu obiektów w języku Java:

1. Tablica przypisuje indeksy liczbowe do obiektów. Przechowuje obiekty znanego typu, nie trzeba więc rzutować wyniku podczas wyszukiwania obiektu. Może być wielowymiarowa i przechowywać typy podstawowe. Jednak jej rozmiar nie może ulec zmianie po stworzeniu.
2. `Collection` przechowuje pojedyncze elementy, podczas gdy odwzorowanie `Map` — pary obiektów skojarzonych. Dzięki typom ogólnym wprowadzonym w Javie SE5 można określać typ obiektów przechowywanych w kontenerach, co blokuje próby wstawiania elementów niewłaściwego typu i eliminuje konieczność rzutowania typów elementów przy wydobywania ich z kontenerów. Tak kolekcje, jak i odwzorowania automatycznie dopasowują swój rozmiar w miarę dodawania elementów. Kontener nie może przechowywać wartości typów podstawowych, ale dzięki mechanizmowi pakowania takich wartości w obiekty posługiwanie się nimi w połączeniu z kontenerami jest całkiem wygodne.
3. Podobnie jak tablica, również lista kojarzy indeksy liczbowe z obiektami — można sobie wyobrazić tablice i listy jako kontenery uporządkowane.
4. Należy stosować `ArrayList`, jeżeli wykonuje się wiele operacji swobodnego dostępu, oraz `LinkedList` — jeżeli wystąpi wiele operacji wstawiania i usuwania ze środka listy.

5. Zachowanie typowe dla kolejek i stosów implementują: `Queue` i `LinkedList`.
6. `Map` jest sposobem na skojarzenie nie liczb, ale *obiektów* z innymi obiektami. Kontener `HashMap` skupia się na szybkim dostępie, podczas gdy `TreeMap` przechowuje swoje klucze w porządku posortowanym, zatem nie jest tak szybki jak `HashMap`. Kontener `LinkedHashMap` przechowuje elementy w kolejności ich dodawania, ale dzięki haszowaniu zapewnia szybki dostęp do elementów.
7. Zbiór `Set` akceptuje tylko jeden egzemplarz każdego z rodzajów obiektu. `HashSet` zapewnia maksymalnie szybkie przeszukiwanie, a `TreeSet` utrzymuje elementy w kolejności posortowanej. Zbiór `LinkedHashSet` przechowuje elementy w kolejności w jakiej były dodawane.
8. Nie ma potrzeby stosowania przestarzałych klas `Vector`, `Hashtable` i `Stack` w nowym kodzie.

Przydatne jest spojrzenie na diagram kontenerów dostępnych w języku Java (z pominięciem klas abstrakcyjych i komponentów przestarzałych). Widać na nim jedynie te interfejsy i klasy, które spotyka się w codziennej praktyce programistycznej.

Taksonomia kontenerów



Jak widać, do czynienia mamy z zaledwie czterema podstawowymi komponentami kontenerów: `Map`, `List`, `Set` i `Queue` — oraz tylko dwoma bądź trzema implementacjami każdego z nich (bez uwzględniania implementacji `Queue` z `java.util.concurrent`). Na powyższym rysunku kontenery, które będą najczęściej wykorzystywane, zostały wyróżnione pogrubieniem.

Prostokąty o liniach kropkowanych reprezentują interfejsy, a rysowane linią pełną — normalne (konkretne) klasy. Strzałki z liniami kropkowanymi oznaczają, że dana klasa implementuje interfejs. Strzałki pełne informują, że klasa może stworzyć obiekty innej klasy, na którą strzałka wskazuje. Przykładowo dowolna klasa `Collection` może powołać do życia `Iterator`, a `List` — `ListIterator` (podobnie jak zwykły `Iterator`, gdyż `List` jest pochodną `Collection`).

Kolejny przykład pokaże różnice pomiędzy metodami pochodzącymi z różnych klas. Kod przykładu pochodzi tak naprawdę z rozdziału „Typy ogólne"; tu jedynie wywołuję go w celu wygenerowania danych do wypisania na wyjściu programu. Wyjście obejmuje też interfejsy implementowane w każdej klasie czy interfejsie:

```
// holding/ContainerMethods.java
import net.mindview.util.*;

public class ContainerMethods {
  public static void main(String[] args) {
    ContainerMethodDifferences.main(args);
  }
} /* Output: (Sample)
Collection: [add, addAll, clear, contains, containsAll, equals, hashCode, isEmpty, iterator, remove,
removeAll, retainAll, size, toArray]
Interfaces in Collection: [Iterable]
Set extends Collection, adds: []
Interfaces in Set: [Collection]
HashSet extends Set, adds: []
Interfaces in HashSet: [Set, Cloneable, Serializable]
LinkedHashSet extends HashSet, adds: []
Interfaces in LinkedHashSet: [Set, Cloneable, Serializable]
TreeSet extends Set, adds: [pollLast, navigableHeadSet, descendingIterator, lower, headSet, ceiling,
pollFirst, subSet, navigableTailSet, comparator, first, floor, last, navigableSubSet, higher, tailSet]
Interfaces in TreeSet: [NavigableSet, Cloneable, Serializable]
List extends Collection, adds: [listIterator, indexOf, get, subList, set, lastIndexOf]
Interfaces in List: [Collection]
ArrayList extends List, adds: [ensureCapacity, trimToSize]
Interfaces in ArrayList: [List, RandomAccess, Cloneable, Serializable]
LinkedList extends List, adds: [pollLast, offer, descendingIterator, addFirst, peekLast, removeFirst,
peekFirst, removeLast, getLast, pollFirst, pop, poll, addLast, removeFirstOccurrence, getFirst, element,
peek, offerLast, push, offerFirst, removeLastOccurrence]
Interfaces in LinkedList: [List, Deque, Cloneable, Serializable]
Queue extends Collection, adds: [offer, element, peek, poll]
Interfaces in Queue: [Collection]
PriorityQueue extends Queue, adds: [comparator]
Interfaces in PriorityQueue: [Serializable]
Map: [clear, containsKey, containsValue, entrySet, equals, get, hashCode, isEmpty, keySet, put, putAll,
remove, size, values]
HashMap extends Map, adds: []
Interfaces in HashMap: [Map, Cloneable, Serializable]
LinkedHashMap extends HashMap, adds: []
Interfaces in LinkedHashMap: [Map]
SortedMap extends Map, adds: [subMap, comparator, firstKey, lastKey, headMap, tailMap]
Interfaces in SortedMap: [Map]
TreeMap extends Map, adds: [descendingEntrySet, subMap, pollLastEntry, lastKey, floorEntry, lastEntry,
lowerKey, navigableHeadMap, navigableTailMap, descendingKeySet, tailMap, ceilingEntry, higherKey,
pollFirstEntry, comparator, firstKey, floorKey, higherEntry, firstEntry, navigableSubMap, headMap,
lowerEntry, ceilingKey]
Interfaces in TreeMap: [NavigableMap, Cloneable, Serializable]
*///:~
```

Najwyraźniej wszystkie zbiory (`Set`) poza `TreeSet` udostępniają identyczny interfejs jak `Collection`. `List` różni się znacznie od `Collection`, choć wymaga metod obecnych w `Collection`. Z drugiej strony, metody w `Queue` stanowią samodzielny zestaw; działająca implementacja `Queue` nie wymaga metod interfejsu `Collection`. Wreszcie jedyną częścią wspólną `Map` i `Collection` jest fakt, że `Map` może generować kolekcje `Collection` za pośrednictwem metod `entrySet()` i `values()`.

Zwróć uwagę na interfejs `java.util.RandomAccess` obecny w `ArrayList`, ale niedostępny w `LinkedList`. To istotna informacja dla algorytmów, które miałyby dynamicznie zmieniać zachowanie w zależności od konkretnej implementacji listy, dla której zostaną wywołane.

Trzeba przyznać, że organizacja biblioteki kontenerów jest cokolwiek zawiła, ale tak bywa w hierarchiach obiektowych. W miarę oswajania się z kontenerami z biblioteki `java.util` (zwłaszcza w ramach lektury rozdziału „Kontenery z bliska") przekonasz się, że układ hierarchii to bynajmniej nie największy problem. Biblioteki kontenerów zawsze były problematycznymi projektami — projekt musi bowiem godzić szereg sprzecznych niekiedy założeń. Nic dziwnego, że tu i ówdzie trzeba było pójść na kompromis.

Mimo to kontenery w Javie stanowią nieocenione narzędzia stosowane na co dzień do upraszczania programów, zwiększania ich wydajności i efektywności. Przywyknięcie do niektórych aspektów biblioteki wymaga czasu, ale sądzę, że zdołasz przemóc początkowe opory, aby potem z wielkim pożytkiem i coraz chętniej wykorzystywać klasy z biblioteki kontenerów.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 12.

Obsługa błędów za pomocą wyjątków

Podstawą ideologii Javy jest założenie, że „źle sformułowany kod nie zostanie wykonany".

Najlepszym momentem na wyłapanie błędu jest kompilacja jeszcze przed próbą uruchomienia programu. Jednak nie wszystkie błędy można wykryć podczas kompilacji. Pozostałe muszą być obsłużone podczas wykonania programu przez jakiś mechanizm pozwalający sprawcy błędu przekazać odpowiednią informację do odbiorcy, który będzie wiedział, jak ma rozwiązać zaistniały problem.

Poprawiona obsługa sytuacji wyjątkowych jest jednym z kluczowych sposobów zwiększenia niezawodności własnego kodu. Obsługa sytuacji wyjątkowych jest zasadniczą sprawą dla każdego pisanego programu, ale jest szczególnie ważna w Javie, ponieważ jednym z jej podstawowych celów jest tworzenie komponentów, które mają być używane przez innych. *Aby system był niezawodny, każdy jego komponent musi być niezawodny.* Zapewniając konsekwentny model informowania o błędach bazujący na wykorzystaniu wyjątków, Java daje komponentom możliwość niezawodnego zgłaszania informacji o problemach do kodu, w którym komponenty te są używane.

Zadaniem mechanizmu obsługi sytuacji wyjątkowych w Javie jest uproszczenie tworzenia dużych, niezawodnych programów przy użyciu mniejszej ilości kodu, niż jest to możliwe obecnie, i przy większej pewności, że w aplikacji nie wystąpi żaden nieobsłużony błąd. Wyjątki nie są bardzo trudne do nauczenia i są jedną z tych funkcji, które dostarczają projektowi wielu natychmiastowych i znaczących korzyści.

Obsługa wyjątków jest jedynym oficjalnym sposobem informowania o błędach w języku Java, a co więcej, jej stosowanie jest wymuszane przez kompilator. Dlatego też nie mogłem napisać dla tej książki więcej przykładów, nie opowiedziawszy wcześniej o wyjątkach. Ten rozdział przedstawia kod potrzebny do prawidłowej obsługi wyjątków oraz sposób, w jaki można tworzyć wyjątki, kiedy pojawią się nowe problemy w tworzonej metodzie.

Zarys koncepcji

W C i innych starszych językach stosowano kilka takich mechanizmów, jednak nie były one częścią języka, tylko zostały ustanowione przez konwencję. Najczęściej polegały na zwróceniu specjalnej wartości lub ustawieniu znacznika. Odbiorca musiał sprawdzać wartość lub znacznik, aby stwierdzić, czy coś poszło nie tak, jak powinno. Jednak z upływem lat odkryto, że programiści, którzy używają takich bibliotek, mają skłonność do megalomanii: „Tak, takie błędy mogą zdarzać się innym, ale nigdy w *moim* kodzie". Nic dziwnego więc, że zaprzestali sprawdzania sytuacji wyjątkowych (a zdarzało się tak, że sytuacje wyjątkowe były zbyt banalne, żeby ktoś nawet pomyślał, by je sprawdzać)[1]. Z drugiej strony, jeśli ktoś *był* na tyle dokładny, by sprawdzać wystąpienie błędu przy każdym wywołaniu metody, to jego kod szybko zmieniał się w nieczytelny koszmar. Pomimo tego programiści byli w stanie sklecić systemy w tych językach, długo więc nie dopuszczali do siebie prawdy — taki sposób obsługi błędów był głównym ograniczeniem przy tworzeniu dużych, wydajnych, dających się pielęgnować programów.

Rozwiązaniem tego problemu jest odrzucenie przypadkowości w obsługiwaniu błędów i wymuszenie pewnych zachowań. Pomysł ten ma już długą historię, ponieważ próby implementacji *obsługi sytuacji wyjątkowych* (ang. *exception handling*) sięgają systemów operacyjnych z lat 60. czy nawet znanego z BASIC-a `on error goto`. Ale obsługa wyjątków w C++ była oparta na Adzie, a Java opiera się głównie na C++ (chociaż przypomina raczej Object Pascal).

Słowo „wyjątek" występuje tu w znaczeniu: „Przyjmuję możliwość wystąpienia wyjątku od tego". W momencie wystąpienia problemu możemy nie wiedzieć, co z nim zrobić, ale wiemy, że nie można go beztrosko zignorować. Trzeba się zatrzymać, a ktoś gdzieś musi wymyślić, co robić dalej. W danym kontekście brakuje informacji potrzebnych do rozwiązania problemu. Zatem przekazujemy problem do szerszego kontekstu, gdzie znajdzie się ktoś z kwalifikacjami odpowiednimi do podjęcia odpowiedniej decyzji.

Inną znaczącą zaletą wyjątków jest to, że zazwyczaj zmniejszają złożoność kodu obsługi błędów. Brak wyjątków wymusza sprawdzanie każdego potencjalnego błędu i jego obsługiwanie w wielu miejscach programu. Dzięki wyjątkom nie trzeba przeprowadzać testu w miejscu wywołania metody (ponieważ wyjątek zagwarantuje, że ktoś go przechwyci). Wystarczy, że obsłuży się problem tylko w jednym miejscu, tak zwanej *procedurze obsługi wyjątku* (ang. *exception handler*)[2]. Zmniejsza to ilość kodu i jednocześnie oddziela kod, który opisuje rozwiązywany problem, od wykonywanego wtedy, kiedy coś pójdzie źle. Czytanie, pisanie i usuwanie błędów w kodzie, w którym używa się wyjątków, jest dużo łatwiejsze od korzystania ze starych metod obsługi błędów.

---

[1] Programista C może sobie obejrzeć wartość zwracaną z `printf()` jako przykład.

[2] Procedury w tym przypadku nie należy rozumieć dosłownie jako wydzielonej funkcji, lecz jako fragment kodu — *przyp. tłum.*

Podstawy obsługi wyjątków

Sytuacja wyjątkowa to problem, który wstrzymuje wykonywanie metody lub bloku. Ważne jest, by oddzielić sytuację wyjątkową od zwykłego problemu — kiedy w aktualnym kontekście istnieje dość przesłanek, by jakoś poradzić sobie z trudnościami. W sytuacji wyjątkowej nie można kontynuować przetwarzania, ponieważ *w aktualnym kontekście* nie ma dostępu do informacji koniecznej do rozwiązania problemu. Wszystko, co można zrobić, to wyjść z aktualnego kontekstu i przekazać problem dalej. Tak się właśnie dzieje, kiedy zgłaszany jest wyjątek.

Prostym przykładem jest dzielenie. Jeśli próbujemy dzielić przez zero, to warto się upewnić, że takie dzielenie nie nastąpi. Ale co oznacza zerowy mianownik przy danym zastosowaniu naszej metody? Być może, w kontekście danego problemu, wiadomo, co zrobić z zerowym mianownikiem. Jeśli jednak jest to wartość nieoczekiwana, nie można z nią niczego zrobić i zamiast iść dalej, trzeba zgłosić wyjątek.

Kiedy zgłaszany jest wyjątek, dzieje się kilka rzeczy. Po pierwsze, w taki sam sposób, jak każdy obiekt Javy, tworzony jest obiekt wyjątku — tworzony na stercie poprzez instrukcję `new`. Aktualna ścieżka wykonania (ta, która nie może być kontynuowana) jest przerywana i obiekt wyjątku jest „wyrzucany" z aktualnego kontekstu. W tym momencie sterowanie przejmuje mechanizm obsługi wyjątków i zaczyna szukać miejsca odpowiedniego do podjęcia dalszego wykonywania programu. Tym odpowiednim miejscem jest *procedura obsługi wyjątku*. Jej zadaniem jest wybrnąć z problemu tak, żeby program mógł spróbować innego rozwiązania lub po prostu kontynuować wykonanie, ignorując błąd.

Jako prosty przykład zgłaszania wyjątku rozważmy obiekt o nazwie `t`. Może się zdarzyć, że otrzymamy uchwyt, który nie został zainicjowany, więc chcielibyśmy to sprawdzić przed wywołaniem metody używającej tego uchwytu. Można przesłać informację o błędzie do szerszego kontekstu przez stworzenie obiektu reprezentującego wiadomość i „wyrzucenie" go na zewnątrz aktualnego kontekstu. Nazywa się to *zgłoszeniem (wyrzuceniem) wyjątku* (ang. *throwing*), wygląda natomiast następująco:

```
if(t == null)
  throw new NullPointerException();
```

Zostanie zgłoszony wyjątek, co w aktualnym kontekście zwalnia nas od odpowiedzialności. Jest on „magicznie" obsługiwany gdzieś indziej. *Gdzie* konkretnie — pokażę niebawem.

Wyjątki pozwalają na traktowanie wszelkich podejmowanych operacji jako transakcji zabezpieczanych przez wyjątki: „zasadniczą przesłanką transakcji jest potrzeba obsługi wyjątków w obliczeniach rozproszonych. Transakcje są komputerowymi odpowiednikami kodeksu kontraktów. Jeśli coś pójdzie nie tak, całe obliczenie uznajemy za niebyłe"[3]. Wyjątki można też traktować jako wbudowany mechanizm wycofywania operacji, bo (przy odrobinie staranności) umożliwiają ustalanie w programie różnych punktów odtworzenia stanu. Jeśli część programu zawiedzie, wyjątek pozwala na wycofanie sterowania do ustalonego stabilnego punktu programu.

[3] Jim Gray (zdobywca nagrody Turinga za wkład w transakcje) w wywiadzie dla *www.acmqueue.org*.

Jeden z ważniejszych aspektów wyjątków objawia się w tym, że w razie niepowodzenia operacji uniemożliwiają one kontynuowanie wykonania programu wedle pierwotnego, zwykłego planu. W językach takich jak C i C++ był to poważny problem; zwłaszcza w C, gdzie nie było możliwości wymuszenia przerwania wykonania zwykłej ścieżki programu w przypadku wykrycia problemu — programista mógł jeszcze długo ignorować błąd, zapędzając program do już zupełnie niepoprawnego stanu. Wyjątki pozwalają więc choćby na wymuszanie przerwania wykonania programu i przyjęcie do wiadomości informacji o błędzie, a w przypadku idealnym na wymuszanie obsłużenia problemu i przywrócenia stabilnego stanu programu.

Argumenty wyjątków

Jak każdy obiekt Javy, wyjątek tworzony jest na stercie przez instrukcję new, która przydziela pamięć i wywołuje konstruktor. We wszystkich standardowych wyjątkach istnieją dwa konstruktory: pierwszy jest konstruktorem domyślnym, a drugi przyjmuje łańcuch jako parametr tak, że w wyjątku można umieścić własny komentarz:

```
if(t == null)
  throw new NullPointerException("t = null");
```

Jak zobaczymy dalej, można później wydobyć ten łańcuch na wiele sposobów.

Słowo kluczowe throw daje kilka ciekawych efektów. Po użyciu new do utworzenia obiektu wyjątku otrzymany uchwyt obiektu należy przekazać do throw. W rezultacie dochodzi do zwrócenia obiektu wyjątku przez metodę nawet wtedy, gdy typ tego obiektu nie jest taki sam, jak zadeklarowany typ zwracany z metody. W uproszczeniu można obsługę wyjątków traktować jako alternatywną metodę zwracania wartości z bloku, chociaż zbyt dosłowne traktowanie tej analogii prowadzi do nieporozumień. Również ze zwykłego bloku można wyjść, zgłaszając wyjątek. Słowem, wyrzucenie wyjątku powoduje zwrócenie obiektu wyjątku i opuszczenie bieżącego zasięgu bądź metody.

Tutaj kończy się podobieństwo do zwykłego powrotu z metody, ponieważ miejsce, *gdzie* następuje powrót, jest zupełnie inne od miejsca, do którego wraca się po normalnym wykonaniu metody (można się znaleźć w procedurze obsługi wyjątku, która będzie dużo dalej — tj. wiele poziomów niżej na stosie wywołań — od miejsca zgłoszenia wyjątku).

Dodatkowo można zgłosić każdy rodzaj obiektu, który można wyrzucić, tj. dziedziczący po klasie Throwable (jest to główna klasa hierarchii wyjątków). Normalnie zgłasza się inną klasę wyjątku dla każdego typu błędu. Informacja o błędzie jest reprezentowana zarówno wewnątrz obiektu wyjątku, jak i wprost przez typ wybranego obiektu wyjątku. Ktoś może się zatem domyślić, co zrobić z otrzymanym wyjątkiem (często jedyną informacją jest typ obiektu wyjątku i żadna informacja mająca znaczenie nie jest przechowywana wewnątrz obiektu wyjątku).

Przechwytywanie wyjątku

Aby zobaczyć, jak wyjątek jest przechwytywany, najpierw musimy zapoznać się z pojęciem *obszaru chronionego* (ang. *guarded region*), który jest fragmentem kodu mogącym zgłaszać wyjątki. Za nim umieszczany jest kod obsługujący te wyjątki.

Blok try

Jeśli wyrzucimy wyjątek, znajdując się wewnątrz metody (albo jedna z wywoływanych wewnątrz metod wyrzuci wyjątek), to wykonanie metody zostanie przerwane przez proces zwracania wyjątku. Jeśli nie chcemy, aby `throw` powodowało wyjście z metody, należy w tej metodzie wstawić specjalny blok przechwytujący wyjątki. Nazywa się on *blokiem prób* (ang. *try*), ponieważ wewnątrz niego „próbujemy" wywołać różne metody. Blok prób jest normalnym blokiem poprzedzonym słowem kluczowym `try`:

```
try {
  // Kod, który może spowodować zgłoszenie wyjątku
}
```

Obsługa błędów w językach programowania nieposiadających mechanizmu obsługi wyjątków wymagała otoczenia każdego wywołania metody kodem ustawiającym i testującym kody błędów, nawet jeśli ta sama metoda była wywoływana kilkakrotnie. Z obsługą wyjątków można wstawić wszystko w blok `try` i przechwytywać wszystkie wyjątki w jednym miejscu. Oznacza to, że kod można łatwiej pisać i łatwiej czytać, ponieważ jego przeznaczenie nie jest przesłonięte przez obsługę błędów.

Obsługa wyjątków

Oczywiście każdy wyrzucony wyjątek musi się gdzieś znaleźć. Tym „miejscem" jest *procedura obsługi wyjątku* i dla każdego typu wyjątków, który chcemy obsłużyć, musi istnieć osobna procedura. Procedury obsługi wyjątków następują bezpośrednio po bloku `try` i są oznaczone przez słowo kluczowe `catch`:

```
try {
  // Kod, który może spowodować zgłoszenie wyjątku
} catch(Type1 id1) {
  // Obsłuż wyjątek typu Type1
} catch(Type2 id2) {
  // Obsłuż wyjątek typu Type2
} catch(Type3 id3) {
  // Obsłuż wyjątek typu Type3
}

// itd...
```

Każdy człon `catch` (procedura obsługi wyjątku) działa jak mała metoda, która pobiera tylko jeden parametr konkretnego typu. Identyfikatory (`id1`, `id2` itd.) mogą być używane wewnątrz procedury tak samo jak parametry metody. Czasami nigdy nie używa się identyfikatorów, ponieważ sam typ wyjątku dostarcza wystarczającą ilość informacji, aby poradzić sobie z wyjątkiem. Mimo to zawsze trzeba deklarować identyfikator.

Procedury obsługi muszą następować bezpośrednio po bloku try. Jeśli zostanie wy cony wyjątek, to mechanizm obsługi wyjątków rozpoczyna poszukiwanie pierwszej cedury, której parametr odpowiada typowi wyrzuconego wyjątku. Wyjątek uważa si obsłużony, gdy wejdzie w sekcję catch. Poszukiwanie procedur zostaje zatrzyman wyjściu z sekcji catch. Wykonywany jest tylko kod pasującej sekcji catch, a nie jak w strukcji switch, która wymaga break na końcu każdego przypadku case, aby zapob wykonywaniu kolejnych przypadków.

Zauważ, że wewnątrz bloku try dowolna liczba różnych wywoływanych metod mo generować ten sam wyjątek, ale wystarczy tylko jedna procedura obsługi do ich obsłużen

Przerwanie czy wznowienie

W teorii obsługi wyjątków wyróżnia się dwa modele. Java obsługuje *przerywanie* (an *termination*)[4], przy którym zakłada się, że jeśli błąd jest krytyczny, nie ma możliwoś powrotu do miejsca, w którym pojawił się wyjątek. Ktokolwiek wyrzucił wyjątek, zd cydował, że nie ma sposobu, aby uratować sytuację i *nie życzy* sobie, aby wracać do teg miejsca.

Alternatywne rozwiązanie nazywa się *wznawianiem*. Oznacza to, że spodziewamy się że procedura obsługi wyjątku zrobi coś, by naprawić system, a następnie można próbowa wywołać wadliwą metodę, zakładając powodzenie drugiej próby. Jeśli chcemy wznowienia, to ciągle mamy nadzieję na kontynuację wykonania po obsłużeniu wyjątku. W tym przypadku wyjątek przypomina raczej wywołanie metody.

Aby zasymulować w Javie mechanizm wznawiania, należałoby powstrzymać się od zgłaszania wyjątków, a zamiast tego wywoływać metody, które powinny eliminować zastane problemy. Alternatywnie można umieścić blok try wewnątrz pętli while, która będzie powtarzała blok try aż do uzyskania żądanego rezultatu.

W przeszłości programiści używający systemów operacyjnych, które obsługiwały wznawianą obsługę wyjątków, ostatecznie wybierali kod podobny do przerywającego, pomijając wznawianie. Zatem mimo że na początku wznawianie może się wydawać atrakcyjne, to w praktyce nie jest zbyt użyteczne. Podstawowym powodem jest najprawdopodobniej powstająca wtedy zależność fragmentów kodu od siebie — procedura obsługi musi wiedzieć, skąd wyrzucono wyjątek, i zawierać odpowiedni kod specyficzny dla każdego takiego miejsca. Powoduje to, że taki kod trudno jest tworzyć i utrzymywać, szczególnie w przypadku dużych systemów, w których wyjątek może być generowany w wielu miejscach.

Tworzenie własnych wyjątków

Nie jesteśmy zobligowani do używania wyłącznie istniejących wyjątków. Hierarchia wyjątków w Javie nie jest w stanie przewidzieć wszystkich błędów, które programista chciałby zgłaszać, istnieje zatem możliwość tworzenia własnych, reprezentujących specyficzne problemy mogące występować w danej bibliotece.

[4] Jak większość języków programowania, w tym C++, C#, Python, D i inne.

Aby utworzyć własną klasę wyjątków, trzeba dokonać dziedziczenia po istniejącym typie wyjątków, najlepiej takim, który jest zbliżony do tworzonego wyjątku (jednak nie zawsze jest to możliwe). W najprostszym przypadku można pozostawić kompilatorowi stworzenie domyślnego konstruktora, więc stworzenie nowej klasy wymaga niewiele kodu.

```
//: exceptions/InheritingExceptions.java
// Tworzenie własnych klas wyjątków.

class SimpleException extends Exception {}

public class InheritingExceptions {
  public void f() throws SimpleException {
    System.out.println("Wyrzucam wyjątek SimpleException z f()");
    throw new SimpleException();
  }
  public static void main(String[] args) {
    InheritingExceptions sed = new InheritingExceptions();
    try {
      sed.f();
    } catch(SimpleException e) {
      System.out.println("Mamy go!");
    }
  }
} /* Output:
Wyrzucam wyjątek SimpleException z f()
Mamy go!
*///:~
```

Kompilator tworzy domyślny konstruktor, który automatycznie (i w sposób niewidoczny) wywołuje domyślny konstruktor klasy bazowej. Oczywiście w tym przypadku nie dostaniemy konstruktora `SimpleException(String)`, ale w praktyce i tak rzadko się go używa. Jak będzie się można niebawem przekonać, najważniejsza w wyjątkach jest nazwa klasy, więc zazwyczaj wystarczający będzie taki wyjątek, jak ten pokazany powyżej.

W tym przypadku wyniki zostały wypisane na konsolę, gdzie zostaną automatycznie przechwycone i przetestowane przez zautomatyzowany system kontroli wyjścia przeznaczony do sprawdzania poprawności działania programów dołączonych do tej książki. Ale komunikaty o błędach można wypisywać do standardowego strumienia diagnostycznego — `System.err`. Przeważnie jest to lepsze miejsce do wysyłania informacji o błędach niż strumień `System.out`, który może zostać przekierowany. Informacja wysłana na `System.err` nie zostanie przekierowana tak jak `System.out`, więc jest bardziej prawdopodobne, że użytkownik ją zauważy.

Można także utworzyć klasę wyjątku, która posiada konstruktor przyjmujący parametry typu `String`.

```
//: exceptions/FullConstructors.java

class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) { super(msg); }
}

public class FullConstructors {
  public static void f() throws MyException {
```

```
    System.out.println("Wyrzucam wyjątek MyException z f()");
    throw new MyException();
  }
  public static void g() throws MyException {
    System.out.println("Wyrzucam wyjątek MyException z g()");
    throw new MyException("Zapoczątkowany w g()");
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException e) {
      e.printStackTrace(System.out);
    }
    try {
      g();
    } catch(MyException e) {
      e.printStackTrace(System.out);
    }
  }
} /* Output:
Wyrzucam wyjątek MyException z f()
MyException
    at FullConstructors.f(FullConstructors.java:11)
    at FullConstructors.main(FullConstructors.java:19)
Wyrzucam wyjątek MyException z g()
MyException: Zapoczątkowany w g()
    at FullConstructors.g(FullConstructors.java:15)
    at FullConstructors.main(FullConstructors.java:24)
*///:~
```

Dodaliśmy tu niewiele kodu — dodatkowe dwa konstruktory, które definiują sposób, w jaki tworzony jest wyjątek `MyException`. W drugim konstruktorze przy użyciu słowa kluczowego `super` wywoływany jest bezpośrednio konstruktor klasy bazowej, przyjmujący argument typu `String`.

W procedurze obsługi wyjątku wywoływana jest jedna metoda klasy `Throwable` (po której dziedziczy klasa `Exception`) — `printStackTrace()`. Na wyjściu widać, że wyświetla ona informacje o wszystkich metodach, które zostały wywołane, by program mógł dotrzeć do miejsca zgłoszenia wyjątku. Informacje te wypisujemy do strumienia `System.out`, gdzie są automatycznie przechwytywane i wypisywane na konsoli. Jednak po wywołaniu wersji domyślnej:

```
e.printStackTrace();
```

informacje trafiłyby na standardowy strumień diagnostyczny.

Ćwiczenie 1. Utwórz klasę z metodą `main()`, która w bloku `try` wyrzuci obiekt klasy `Exception`. Do konstruktora `Exception` powinieneś przekazać obiekt `String`. Przechwyć wyjątek w bloku `catch` i wypisz na wyjściu argument wyjątku. W klauzuli `finally` wypisz komunikat dowodzący wykonania kodu z tejże klauzuli (2).

Ćwiczenie 2. Zdefiniuj referencję do obiektu i zainicjalizuj ją wartością pustą (`null`). Spróbuj, używając tej referencji, wywołać metodę. Następnie opakuj ten kod w blok `try-catch`, aby przechwycić wyjątek (1).

Ćwiczenie 3. Napisz kod generujący i przechwytujący wyjątek typu `ArrayIndexOutOfBoundsException` (indeks tablicy poza zakresem) (1).

Ćwiczenie 4. Utwórz własną klasę wyjątków, używając słowa kluczowego `extends`. Napisz dla tej klasy konstruktor przyjmujący parametr typu `String` i zapamiętujący ten parametr wewnątrz obiektu. Napisz metodę, która wyświetla zapamiętany łańcuch. Utwórz blok `try-catch`, aby wypróbować nowy wyjątek (2).

Ćwiczenie 5. Stwórz własny kod wznawiający wykonanie, używając pętli `while`, która jest powtarzana, dopóki wyjątek nie przestanie być zgłaszany (3).

Rejestrowanie wyjątków

Wyjątki można także *rejestrować* (*logować*) przy użyciu mechanizmów z biblioteki `java.util.logging`. Pełne omówienie zagadnienia rejestrowania znajduje się w suplemencie publikowanym pod adresem *http://MindView.net/Books/BetterJava*; nam wystarczy na razie opanowanie podstawowych technik rejestrowania.

```
//: exceptions/LoggingExceptions.java
// Wyjątek zgłaszający błąd za pośrednictwem rejestratora (Logger).
import java.util.logging.*;
import java.io.*;

class LoggingException extends Exception {
  private static Logger logger =
    Logger.getLogger("LoggingException");
  public LoggingException() {
    StringWriter trace = new StringWriter();
    printStackTrace(new PrintWriter(trace));
    logger.severe(trace.toString());
  }
}

public class LoggingExceptions {
  public static void main(String[] args) {
    try {
      throw new LoggingException();
    } catch(LoggingException e) {
      System.err.println("Przechwycono " + e);
    }
    try {
      throw new LoggingException();
    } catch(LoggingException e) {
      System.err.println("Przechwycono " + e);
    }
  }
} /* Output: (85% match)
Aug 30, 2005 4:02:31 PM LoggingException <init>
SEVERE: LoggingException
    at LoggingExceptions.main(LoggingExceptions.java:19)

Przechwycono LoggingException
Aug 30, 2005 4:02:31 PM LoggingException <init>
SEVERE: LoggingException
    at LoggingExceptions.main(LoggingExceptions.java:24)

Przechwycono LoggingException
*///:~
```

Statyczna metoda `Logger.getLogger()` zwraca obiekt rejestratora (`Logger`) skojarzony z argumentem typu `String` (zwykle reprezentuje on nazwę pakietu i klasy, której dotyczy rejestrowanie) wypisującym informacje na standardowym wyjściu diagnostycznym (`System.err`). Najprostszym sposobem pisania do logu jest wywołanie metody odpowiadającej poziomowi rejestrowanego komunikatu — tu używamy metody `severe()`. W komunikacie chcielibyśmy ująć stos wywołań, które doprowadziły do wyjątku — szkoda, że `printStackTrace()` nie zwraca obiektu `String`. Aby otrzymać taki obiekt, musimy użyć przeciążonej wersji `printStackTrace()` przyjmującej argument w postaci obiektu klasy `java.io.PrintWriter` (wszystko wyjaśni się w rozdziale „Wejście-wyjście"). Jeśli konstruktor `PrintWriter` zasilimy argumentem `java.io.StringWriter`, otrzymamy wartość, z której metodą `toString()` będzie można wyłuskać obiekt `String`.

Choć podejście zaprezentowane w klasie `LoggingException` jest bardzo wygodne — cała infrastruktura rejestracji jest wbudowywana w wyjątek, dzięki czemu całość działa bez interwencji ze strony programisty-klienta — najczęściej przychodzi nam przechwytywać i rejestrować cudze klasy wyjątków, co wymaga wygenerowania wpisu do logu w ramach procedury obsługi wyjątku:

```
//: exceptions/LoggingExceptions2.java
// Rejestrowanie przechwyconych wyjątków.
import java.util.logging.*;
import java.io.*;

public class LoggingExceptions2 {
  private static Logger logger =
    Logger.getLogger("LoggingExceptions2");
  static void logException(Exception e) {
    StringWriter trace = new StringWriter();
    e.printStackTrace(new PrintWriter(trace));
    logger.severe(trace.toString());
  }
  public static void main(String[] args) {
    try {
      throw new NullPointerException();
    } catch(NullPointerException e) {
      logException(e);
    }
  }
} /* Output: (90% match)
2006-04-04 18:43:29 LoggingExceptions2 logException
SEVERE: java.lang.NullPointerException
    at LoggingExceptions2.main(LoggingExceptions2.java:16)
*///:~
```

Proces tworzenia własnych wyjątków można rozwinąć jeszcze bardziej. Można bowiem dodać własne konstruktory i składowe klasy:

```
//: exceptions/ExtraFeatures.java
// Dalsze polerowanie własnych klas wyjątków.
import static net.mindview.util.Print.*;

class MyException2 extends Exception {
  private int x;
  public MyException2() {}
  public MyException2(String msg) { super(msg); }
```

```
  public MyException2(String msg, int x) {
    super(msg);
    this.x = x;
  }
  public int val() { return x; }
  public String getMessage() {
    return "Komunikat szczegółowy: "+ x + " "+ super.getMessage();
  }
}

public class ExtraFeatures {
  public static void f() throws MyException2 {
    print("Wyrzucam wyjątek MyException2 z f()");
    throw new MyException2();
  }
  public static void g() throws MyException2 {
    print("Wyrzucam wyjątek MyException2 z g()");
    throw new MyException2("Zapoczątkowany w g()");
  }
  public static void h() throws MyException2 {
    print("Wyrzucam wyjątek MyException2 z h()");
    throw new MyException2("Zapoczątkowany w h()", 47);
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException2 e) {
      e.printStackTrace(System.out);
    }
    try {
      g();
    } catch(MyException2 e) {
      e.printStackTrace(System.out);
    }
    try {
      h();
    } catch(MyException2 e) {
      e.printStackTrace(System.out);
      System.out.println("e.val() = " + e.val());
    }
  }
} /* Output:
Wyrzucam wyjątek MyException2 z f()
MyException2: Komunikat szczegółowy: 0 null
        at ExtraFeatures.f(ExtraFeatures.java:22)
        at ExtraFeatures.main(ExtraFeatures.java:34)
Wyrzucam wyjątek MyException2 z g()
MyException2: Komunikat szczegółowy: 0 Zapoczątkowany w g()
        at ExtraFeatures.g(ExtraFeatures.java:26)
        at ExtraFeatures.main(ExtraFeatures.java:39)
Wyrzucam wyjątek MyException2 z h()
MyException2: Komunikat szczegółowy: 47 Zapoczątkowany w h()
        at ExtraFeatures.h(ExtraFeatures.java:30)
        at ExtraFeatures.main(ExtraFeatures.java:44)
e.val() = 47
*///:~
```

Dodaliśmy pole x razem z metodą, która odczytuje jego wartość, oraz dodatkowym konstruktorem jc ustawiającym. Dodatkowo metoda `Throwable.getMessage()` została przesłonięta w celu generowania bardziej interesujących i szczegółowych komunikatów. W klasach wyjątków metoda `getMessage()` pełni podobną rolę, jak metoda `toString()` we wszystkich obiektach Javy.

Ponieważ wyjątek to obiekt jak wszystkie inne, proces rozbudowy własnych klas wyjątków można posunąć jeszcze dalej. Jednak należy pamiętać, że cała ta „dekoracja" może zostać pominięta przez programistę wykorzystującego ten pakiet z zewnątrz, gdyż może on jedynie sprawdzać, czy w metodzie wyrzucono wyjątek i nic poza tym (w ten sposób używa się większości wyjątków z biblioteki Javy).

Ćwiczenie 6. Utwórz dwie klasy wyjątków, z których każda będzie automatycznie realizowała rejestrację. Zaprezentuj efekty pracy (1).

Ćwiczenie 7. Zmień ćwiczenie 3. tak, aby w bloku `catch` odbywało się rejestrowanie wyjątku (1).

Specyfikacja wyjątków

W Javie wymaganc jest informowanie programistów, wywołujących napisaną przez nas metodę, o wyjątkach, jakie mogą zostać przez nią zgłoszone. Jest to dobra zasada, ponieważ dzięki temu osoba wywołująca wie, co musi napisać, aby przechwycić wszystkie możliwe wyjątki. Oczywiście jeśli dostępny jest kod źródłowy, można go po prostu przejrzeć w poszukiwaniu instrukcji `throw`. Najczęściej jednak źródła bibliotek nie są dostarczane. Aby zapobiec tego typu problemom, Java dostarcza składnię (oraz *wymusza* jej stosowanie) umożliwiającą uprzejme informowanie innego programisty, jakie wyjątki dana metoda może wyrzucić, przez co programista jest w stanie je obsłużyć. Nazywa się to *specyfikacją wyjątków* i jest częścią deklaracji metody, pojawiającą się po liście parametrów.

Specyfikacja wyjątków wykorzystuje dodatkowe słowo kluczowe `throws`, po którym następuje lista wszystkich potencjalnych typów wyjątków. Przykładowa definicja metody może wyglądać tak:

```
void f() throws TooBig, TooSmall, DivZero { // ...
```

Jeśli napiszemy:

```
void f() { // ...
```

oznacza to, że żadne wyjątki nie są wyrzucane z tej metody (*oprócz* wyjątków typu `RuntimeException`, które mogą się pojawić praktycznie wszędzie i to bez żadnych specyfikacji — opiszę to dalej).

Nie można oszukać specyfikacji wyjątków — jeśli metoda powoduje wyjątki i nie obsługuje ich, kompilator wykryje to i zgłosi, że należy albo obsłużyć wyjątek, albo zaznaczyć w specyfikacji wyjątków, że ten wyjątek może być wyrzucony z metody. Egzekwując specyfikację wyjątków od góry do dołu, Java gwarantuje, że poprawność wyjątków może być zapewniona w *czasie kompilacji*.

W jednym miejscu można skłamać — można twierdzić, że wyrzuca się wyjątek, a w rzeczywistości tego nie robić. Kompilator wierzy nam na słowo i zmusza użytkownika naszej metody, aby potraktował ją tak, jakby rzeczywiście wyrzucała taki wyjątek. Dzięki temu można sobie „zaklepać miejsce na później". Na przykład jeśli później tego typu wyjątki będą wyrzucane, nie będzie to wymagało zmian w kodzie wywołującym metodę. Jest to również ważne przy tworzeniu klas *abstrakcyjnych* oraz *interfejsów*, których klasy pochodne lub implementacje mogą wymagać możliwości zgłaszania wyjątków.

Wyjątki sprawdzane przez kompilator są nazywane *wyjątkami sprawdzanymi* (ang. *checked exceptions*).

Ćwiczenie 8. Napisz klasę z metodą, która zgłasza wyjątek stworzony w ćwiczeniu 4. Spróbuj go skompilować bez specyfikacji wyjątku, aby zobaczyć, co zrobi kompilator. Dodaj odpowiednią specyfikację wyjątku. Wypróbuj swoją klasę i jej wyjątki wewnątrz bloku `try-catch` (1).

Przechwytywanie dowolnego wyjątku

Możliwe jest stworzenie procedury obsługi przechwytującej wyjątki dowolnego typu. Można to zrobić przez przechwytywanie wyjątków klasy podstawowej `Exception` (są jeszcze inne typy podstawowych wyjątków, ale `Exception` jest klasą bazową, której można użyć do obsługi niemal wszystkich sytuacji programistycznych).

```
catch(Exception e) {
  System.err.println("Złapałem wyjątek");
}
```

Spowoduje to przechwycenie każdego wyjątku. Zatem jeśli używamy takiej procedury w połączeniu z procedurami obsługi innych wyjątków, ważne jest, by umieścić ją na *końcu* listy procedur, gdyż w przeciwnym razie pozostałe procedury zostaną pominięte.

Klasa `Exception` jest klasą bazową dla wszystkich klas wyjątków, jakie mogą być istotne dla programisty. Dlatego nie przekazuje ona zbyt szczegółowych informacji o wyjątku. Można jednak wywołać metody pochodzące z *jej* klasy bazowej, czyli `Throwable`:

```
String getMessage()
String getLocalizedMessage()
```

Zwracają szczegółowy komunikat lub komunikat lokalizowany.

```
String toString()
```

Zwraca krótki opis `Throwable` łącznie z wiadomością o szczegółach, jeśli jest dostępna.

```
void printStackTrace()
void printStackTrace(java.io.PrintStream)
void printStackTrace(java.io.PrintWriter)
```

Wypisują `Throwable` i ślad stosu wywołań. Stos wywołań pokazuje sekwencję wywołań metod prowadzącą do miejsca, w którym został wyrzucony wyjątek. Pierwsza wersja drukuje na standardowe wyjście diagnostyczne, druga i trzecia — do podanego

strumienia (w rozdziale „Wejście-wyjście" zostanie wyjaśnione, dlaczego używa się dwóch typów strumieni).

```
Throwable fillInStackTrace()
```

Zapisuje w obiekcie `Throwable` informacje o aktualnym stanie stosu. Jest to przydatne, kiedy aplikacja ponownie wyrzuca błąd lub wyjątek (więcej na ten temat za moment).

Dodatkowo można użyć innych metod z klasy bazowej dla `Throwable`, którą jest typ `Object` (typ podstawowy dla wszystkich innych). W przypadku wyjątków przydatna może być metoda `getClass()`, która zwraca obiekt reprezentujący klasę danego obiektu. Można wtedy zapytać obiekt typu `Class` o jego nazwę przez metodę `getName()` (zwracającą nazwę klasy wraz z nazwą pakietu) albo `getSimpleName()` (która zwraca jedynie nazwę klasy).

Poniższy przykład pokazuje sposób użycia podstawowych metod klasy `Exception`.

```
//: exceptions/ExceptionMethods.java
// Demonstracja metod wyjątków.
import static net.mindview.util.Print.*;

public class ExceptionMethods {
  public static void main(String[] args) {
    try {
      throw new Exception("Mój wyjątek");
    } catch(Exception e) {
      print("Przechwycono: Exception");
      print("getMessage(): " + e.getMessage());
      print("getLocalizedMessage(): " +
        e.getLocalizedMessage());
      print("toString(): " + e);
      print("printStackTrace(): ");
      e.printStackTrace(System.out);
    }
  }
} /* Output:
Przechwycono: Exception
getMessage(): Mój wyjątek
getLocalizedMessage(): Mój wyjątek
toString():java.lang.Exception: Mój wyjątek
printStackTrace():
java.lang.Exception: Mój wyjątek
      at ExceptionMethods.main(ExceptionMethods.java:8)
*///:~
```

Widać, że metody podają coraz więcej informacji — każda kolejna jest efektywnie nadzbiorem poprzedniej.

Ćwiczenie 9. Stwórz trzy nowe typy wyjątków. Napisz klasę z metodą, która zgłasza wszystkie trzy. W metodzie `main()` wywołaj tę metodę, ale użyj pojedynczej sekcji `catch`, która przechwyci wszystkie trzy typy wyjątków (2).

Stos wywołań

Informacje udostępniane przez metodę `printStackTrace()` można pozyskać wprost za pośrednictwem wywołania `getStackTrace()`. Ta druga metoda zwraca tablicę elementów stosu wywołań, z których każdy reprezentuje pojedynczą ramkę stosu. Element zerowy to szczyt stosu, czyli ostatnio zrealizowane wywołanie metody w sekwencji (a więc miejsce, w którym doszło do utworzenia i wyrzucenia obiektu `Throwable`). Ostatni element tablicy to dno stosu, czyli pierwsze wywołanie metody w sekwencji wywołań. Demonstruje to poniższy prosty program:

```
//: exceptions/WhoCalled.java
// Programowy dostęp do informacji o stosie wywołań.

public class WhoCalled {
  static void f() {
    // Wygenerowanie wyjątku w celu wypełnienia stosu wywołań
    try {
      throw new Exception();
    } catch (Exception e) {
      for(StackTraceElement ste : e.getStackTrace())
        System.out.println(ste.getMethodName());
    }
  }
  static void g() { f(); }
  static void h() { g(); }
  public static void main(String[] args) {
    f();
    System.out.println("--------------------------------");
    g();
    System.out.println("--------------------------------");
    h();
  }
} /* Output:
f
main
--------------------------------
f
g
main
--------------------------------
f
g
h
main
*///:~
```

Ograniczyliśmy się tu do wypisania nazw wywoływanych metod, ale moglibyśmy równie dobrze wypisywać w całości elementy `StackTraceElement` zawierające mnóstwo dodatkowych informacji.

Ponowne wyrzucanie wyjątków

Niejednokrotnie trzeba wyrzucić ponownie wyjątek, który został dopiero co przechwycony, szczególnie jeśli używa się `Exception`, by przechwycić wszystkie wyjątki. Ponieważ posiadamy już referencję do aktualnego wyjątku, można po prostu ponownie wyrzucić tę referencję.

```
catch(Exception e) {
  System.err.println("Zgłoszono wyjątek");
  throw e;
}
```

Ponowne wyrzucenie wyjątku powoduje przekazanie wyjątku do procedur obsługi na kolejnym poziomie. Wszystkie pozostałe klauzule `catch` w tym samym bloku `try` są pomijane. Dodatkowo zachowywane są wszystkie informacje dotyczące obiektu wyjątku. Tak więc procedura obsługi na wyższym poziomie, przechwytująca wyjątki danego typu, może odzyskać z tego obiektu wszelkie informacje.

Jeśli po prostu wyrzuci się aktualny wyjątek, to informacja o wyjątku, wyświetlona przez `printStackTrace()`, będzie odnosić się do źródła wyjątku, a nie do miejsca, w którym był on ponownie wyrzucony. Jeśli chcemy podać nową informację o śladzie stosu, można wywołać metodę `fillInStackTrace()`, która zwraca obiekt wyjątku utworzony przez zamieszczenie aktualnej informacji o stosie w starym wyjątku. Oto jak wygląda:

```
//: exceptions/Rethrowing.java
// Demonstracja metody fillInStackTrace()

public class Rethrowing {
  public static void f() throws Exception {
    System.out.println("wyjątek zapoczątkowany w f()");
    throw new Exception("wyrzucony z f()");
  }
  public static void g() throws Exception {
    try {
      f();
    } catch(Exception e) {
      System.out.println("W g(), e.printStackTrace()");
      e.printStackTrace(System.out);
      throw e;
    }
  }
  public static void h() throws Exception {
    try {
      f();
    } catch(Exception e) {
      System.out.println("W h(), e.printStackTrace()");
      e.printStackTrace(System.out);
      throw (Exception)e.fillInStackTrace();
    }
  }
  public static void main(String[] args) {
    try {
      g();
    } catch(Exception e) {
      System.out.println("Metoda main: printStackTrace()");
      e.printStackTrace(System.out);
    }
    try {
      h();
    } catch(Exception e) {
      System.out.println("Metoda main: printStackTrace()");
      e.printStackTrace(System.out);
    }
  }
```

```
} /* Output:
wyjątek zapoczątkowany w f()
W g(), e.printStackTrace()
java.lang.Exception: wyrzucony z f()
        at Rethrowing.f(Rethrowing.java:7)
        at Rethrowing.g(Rethrowing.java:11)
        at Rethrowing.main(Rethrowing.java:29)
Metoda main: printStackTrace()
java.lang.Exception: wyrzucony z f()
        at Rethrowing.f(Rethrowing.java:7)
        at Rethrowing.g(Rethrowing.java:11)
        at Rethrowing.main(Rethrowing.java:29)
wyjątek zapoczątkowany w f()
W h(), e.printStackTrace()
java.lang.Exception: wyrzucony z f()
        at Rethrowing.f(Rethrowing.java:7)
        at Rethrowing.h(Rethrowing.java:20)
        at Rethrowing.main(Rethrowing.java:35)
Metoda main: printStackTrace()
java.lang.Exception: wyrzucony z f()
        at Rethrowing.h(Rethrowing.java:24)
        at Rethrowing.main(Rethrowing.java:35)
*///:~
```

Wiersz z wywołaniem `fillInStackTrace()` staje się nowym punktem pochodzenia wyjątku.

Można również wyrzucić inny wyjątek niż ten, który został złapany. Wtedy otrzymuje się podobny efekt, jak przy użyciu `fillInStackTrace()` — informacja o pochodzeniu wyjątku jest gubiona, a pozostaje tylko informacja odnosząca się do nowego wyjątku.

```
//: exceptions/RethrowNew.java
// Wyrzucenie wyjątku innego niż przechwycony.

class OneException extends Exception {
  public OneException(String s) { super(s); }
}

class TwoException extends Exception {
  public TwoException(String s) { super(s); }
}

public class RethrowNew {
  public static void f() throws OneException {
    System.out.println("wyjątek zapoczątkowany w f()");
    throw new OneException("wyrzucony z f()");
  }
  public static void main(String[] args) {
    try {
      try {
        f();
      } catch(OneException e) {
        System.out.println(
          "Przechwycony w wewnętrznym bloku try, e.printStackTrace()");
        e.printStackTrace(System.out);
        throw new TwoException("wyrzucony z wewnętrznego bloku try");
      }
    } catch(TwoException e) {
```

```
        System.out.println(
          "Przechwycony w zewnętrznym bloku try, e.printStackTrace()");
        e.printStackTrace(System.out);
      }
    }
  } /* Output:
wyjątek zapoczątkowany w f()
Przechwycony w wewnętrznym bloku try, e.printStackTrace()
OneException: wyrzucony z f()
        at RethrowNew.f(RethrowNew.java:15)
        at RethrowNew.main(RethrowNew.java:20)
Przechwycony w zewnętrznym bloku try, e.printStackTrace()
TwoException: wyrzucony z wewnętrznego bloku try
        at RethrowNew.main(RethrowNew.java:25)
*///:~
```

Ostateczny wyjątek wie tylko, że został wyrzucony z `main()`, a nie z `f()`.

Nie trzeba przejmować się zwalnianiem pamięci po poprzednim wyjątku, a jeśli już o tym mowa — to zwalnianiem pamięci po jakimkolwiek wyjątku. Są one prawdziwymi obiektami umieszczanymi na stercie przez `new`, więc mechanizm zwalniania pamięci automatycznie je uprzątnie.

Sekwencje wyjątków

Często zdarza się, że chcemy przechwycić jeden wyjątek i zgłosić inny, zachowując przy tym informację o oryginalnym wyjątku. Rozwiązanie takie nazywa się tworzeniem *sekwencji* (lub *łańcucha*) *wyjątków*. Do momentu pojawienia się JDK 1.4 programiści musieli tworzyć własny kod umożliwiający przechowywanie informacji o oryginalnym wyjątku, teraz jednak wszystkie klasy potomne klasy `Throwable` dysponują konstruktorem umożliwiającym przekazanie obiektu *przyczyny* (ang. *cause*). Argument ten jest przeznaczony do przekazywania oryginalnego wyjątku, dzięki czemu, pomimo tworzenia zupełnie nowego wyjątku, zachowywany jest oryginalny zrzut stosu.

Ciekawe jest spostrzeżenie, że jedynymi klasami potomnymi klasy `Throwable` udostępniającymi konstruktory pozwalające na przekazanie obiektu *przyczyny* są trzy podstawowe klasy wyjątków: `Error` (używana przez wirtualną maszynę Javy w celu zgłaszania błędów systemowych), `Exception` oraz `RuntimeException`. Aby utworzyć sekwencję zawierającą dowolny inny typ wyjątku, obiekt przyczyny należy przekazać w wywołaniu metody `initCause()`, a nie w konstruktorze.

Oto przykład pozwalający na dynamicznie dodawanie pól do obiektu `DynamicFields` podczas działania programu:

```
//: exceptions/DynamicFields.java
// Klasa, ktora dynamicznie uzupełnia się polami.
// Demonstracja montowania sekwencji wyjątków.
import static net.mindview.util.Print.*;

class DynamicFieldsException extends Exception {}

public class DynamicFields {
  private Object[][] fields;
```

```
public DynamicFields(int initialSize) {
  fields = new Object[initialSize][2];
  for(int i = 0; i < initialSize; i++)
    fields[i] = new Object[] { null, null };
}
public String toString() {
  StringBuilder result = new StringBuilder();
  for(Object[] obj : fields) {
    result.append(obj[0]);
    result.append(": ");
    result.append(obj[1]);
    result.append("\n");
  }
  return result.toString();
}
private int hasField(String id) {
  for(int i = 0; i < fields.length; i++)
    if(id.equals(fields[i][0]))
      return i;
  return -1;
}
private int
getFieldNumber(String id) throws NoSuchFieldException {
  int fieldNum = hasField(id);
  if(fieldNum == -1)
    throw new NoSuchFieldException();
  return fieldNum;
}
private int makeField(String id) {
  for(int i = 0; i < fields.length; i++)
    if(fields[i][0] == null) {
      fields[i][0] = id;
      return i;
    }
  // Brak pustych pól. Dodanie nowego:
  Object[][] tmp = new Object[fields.length + 1][2];
  for(int i = 0; i < fields.length; i++)
    tmp[i] = fields[i];
  for(int i = fields.length; i < tmp.length; i++)
    tmp[i] = new Object[] { null, null };
  fields = tmp;
  // Rekurencyjne wywołanie z nowymi polami:
  return makeField(id);
}
public Object
getField(String id) throws NoSuchFieldException {
  return fields[getFieldNumber(id)][1];
}
public Object setField(String id, Object value)
throws DynamicFieldsException {
  if(value == null) {
    // Większość wyjątków nie posiada konstruktora "przyczyny".
    // W takich przypadkach trzeba użyć metody initCause(),
    // dostępnej we wszystkich podtypach Throwable.
    DynamicFieldsException dfe =
      new DynamicFieldsException();
```

```
        dfe.initCause(new NullPointerException());
        throw dfe;
      }
      int fieldNumber = hasField(id);
      if(fieldNumber == -1)
        fieldNumber = makeField(id);
      Object result = null;
      try {
        result = getField(id); // Pobranie poprzedniej wartości
      } catch(NoSuchFieldException e) {
        // Użycie konstruktora przyjmującego "przyczynę":
        throw new RuntimeException(e);
      }
      fields[fieldNumber][1] = value;
      return result;
    }
    public static void main(String[] args) {
      DynamicFields df = new DynamicFields(3);
      print(df);
      try {
        df.setField("d", "Wartość dla d");
        df.setField("number", 47);
        df.setField("number2", 48);
        print(df);
        df.setField("d", "Nowa wartość dla d");
        df.setField("number3", 11);
        print("df: " + df);
        print("df.getField(\"d\") : " + df.getField("d"));
        Object field = df.setField("d", null); // Wyjątek
      } catch(NoSuchFieldException e) {
        e.printStackTrace(System.out);
      } catch(DynamicFieldsException e) {
        e.printStackTrace(System.out);
      }
    }
} /* Output:
null: null
null: null
null: null

d: Wartość dla d
number: 47
number2: 48

df: d: Nowa wartość dla d
number: 47
number2: 48
number3: 11

df.getField("d") : Nowa wartość dla d
DynamicFieldsException
        at DynamicFields.setField(DynamicFields.java:65)
        at DynamicFields.main(DynamicFields.java:94)
Caused by: java.lang.NullPointerException
        at DynamicFields.setField(DynamicFields.java:66)
        ... 1 more
*///:~
```

Każdy obiekt DynamicField zawiera tablicę par Object-Object. Pierwszym z elementów tych par jest identyfikator pola (obiekt String), a drugim dowolna wartość z wyjątkiem wartości typów podstawowych. Tworząc obiekt, staramy się określić, ilu pól będziemy potrzebować. W wyniku wywołania metoda setField() bądź odnajduje istniejące pole o podanej nazwie, bądź też tworzy nowe i zapisuje w nim przekazaną wartość. Jeśli pojemność tablicy zostanie wyczerpana, tworzona jest nowa tablica o jeden element większa, a wszystkie pary przechowywane w tablicy oryginalnej zostają do niej przeniesione. W przypadku próby dodania wartości null zgłaszany jest wyjątek DynamicFieldsException — w tym celu tworzony jest obiekt tej klasy, a następnie w wywołaniu metody initCause() zostaje przekazany wyjątek NullPointerException.

Jako wartość wynikową metoda setField() zwraca także dotychczasową wartość pola, używając do tego celu metodę getField(), która może zgłaszać wyjątek NoSuchField-Exception. Jeśli programista wywoła metodę getField() to on będzie odpowiedzialny za obsługę wyjątku NoSuchFieldException; jeśli jednak wyjątek ten zostanie zgłoszony wewnątrz metody setField(), to nie jest on błędem programisty, zostaje zatem skonwertowany do wyjątku RuntimeException przy użyciu konstruktora pobierającego *przyczynę*.

Zwróć uwagę, że metoda toString() tworzy wynik za pomocą obiektu StringBuilder. O obicktach tej klasy dowiemy się więcej w rozdziale „Ciągi znaków"; zasadniczo używa się ich zawsze przy definiowaniu metody toString(), która składa ciąg wynikowy w pętli.

Ćwiczenie 10. Utwórz klasę z dwoma metodami f() i g(). W g() zgłoś wyjątek nowego, zdefiniowanego przez Ciebie typu. W f() wywołaj g(), przechwyć jej wyjątki i w sekcji catch zgłoś inny wyjątek (drugiego zdefiniowanego przez Cicbie typu). Przetestuj swój kod w main() (2).

Ćwiczenie 11. Powtórz powyższe ćwiczenie, lecz tym razem w klauzuli catch umieść wyjątek zgłoszony przez metodę g() wewnątrz wyjątku RuntimeException (1).

Standardowe wyjątki Javy

Klasa Throwable opisuje wszystko, co może być zgłoszone jako wyjątek. Istnieją dwa ogólne rodzaje obiektów Throwable (tutaj „rodzaje" oznacza „dziedziczące po"). Error reprezentuje błędy kompilacji oraz systemu, których przechwytywaniem nie ma potrzeby się przejmować (z wyjątkiem specjalnych przypadków). Exception jest podstawowym typem, jaki może być wyrzucony z dowolnej metody klasy biblioteki standardowej Javy i własnej metody lub w wyniku innych błędów przy wykonaniu. Zatem dla programisty Javy obiektem zainteresowania jest typ Exception.

Najlepszym sposobem na zapoznanie się z wyjątkami jest przeglądanie dokumentacji JDK Javy. Warto to zrobić, aby zapoznać się z różnymi wyjątkami. Jednak wkrótce zobaczymy, że jeden wyjątek nie różni się niczym szczególnym od drugiego, jeśli nie liczyć nazwy. Liczba wyjątków w Javie stale się zwiększa — zasadniczo bezcelowe byłoby wymienianie ich w książce. Dodatkowo każda nowa biblioteka będzie najprawdopodobniej zawierała swoje własne wyjątki. Ważne jest, aby zrozumieć samo pojęcie wyjątku oraz co należy z nimi robić.

Podstawowa zasada polega na tym, że nazwa wyjątku określa problem, jaki wystąpił, oraz ma być relatywnie samoopisująca. Nie wszystkie wyjątki są zdefiniowane w `java.lang`. Niektóre zostały stworzone do obsługi innych bibliotek, takich jak: `util`, `net` i `io`. Można to poznać po pełnych nazwach ich klas lub po tym, po jakiej klasie dziedziczą. Na przykład wszystkie wyjątki wejścia-wyjścia są dziedziczone z `java.io.IOException`.

Przypadek specjalny: RuntimeException

Pierwszym przykładem w tym rozdziale było:

```
if(t == null)
  throw new NullPointerException();
```

Potrzeba sprawdzania, czy referencja nie ma wartości `null` przy każdym wywołaniu metody (ponieważ nie wiadomo, czy metoda przekazała prawidłową referencję), może się wydawać nieco przerażająca. Na szczęście nie ma takiej konieczności — jest to częścią standardowej procedury sprawdzania, którą sama Java przeprowadza w trakcie wykonania. Jeśli nastąpi jakiekolwiek odwołanie do nieprzypisanej referencji, Java automatycznie zgłosi wyjątek `NullPointerException`. Zatem powyższy kod jest zawsze zbyteczny, chyba że chodzi o wykonanie dodatkowych testów dla zabezpieczenia się przed niepożądanym w danym miejscu wyjątkiem `NullPointerException`.

Istnieje cała grupa wyjątków tej kategorii. Są one zgłaszane zawsze automatycznie przez Javę i nie trzeba uwzględniać ich w specyfikacji wyjątków. Dla wygody wszystkie są zgrupowane przez umieszczenie ich pod jedną wspólną klasą bazową o nazwie `RuntimeException`. Jest to doskonały przykład dziedziczenia: ustalana jest rodzina typów, które mają wspólną charakterystykę i zachowanie. Nie ma również potrzeby, żeby kiedykolwiek pisać w specyfikacji wyjątków, że metoda może zgłosić wyjątek `RuntimeException` (lub wyjątek którejkolwiek z klas potomnych), gdyż są to *wyjątki niesprawdzane* (ang. *unchecked exceptions*). Ponieważ oznaczają błąd programisty, praktycznie nigdy nie przechwytuje się wyjątków `RuntimeException` — jest to załatwiane automatycznie. Jeśli bylibyśmy zmuszeni do sprawdzania, czy nie wystąpił wyjątek `RuntimeExceptions`, kod mógłby stać się niechlujny. Mimo że normalnie nie przechwytuje się wyjątków `RuntimeException` we własnym kodzie, można czasem zdecydować się na zgłoszenie któregoś z tych wyjątków.

Co się dzieje, jeśli taki wyjątek nie zostanie przechwycony? Ponieważ kompilator nie wymusza dla nich specyfikacji wyjątków, jest bardzo prawdopodobne, że wyjątek `RuntimeException` przedostanie się aż do metody `main()`, nie będąc przechwycony po drodze. Aby zobaczyć, co się wtedy dzieje, spójrzmy na następujący przykład:

```
// exceptions/NeverCaught.java
// Ignorowanie wyjątków RuntimeException.
// {ThrowsException}

public class NeverCaught {
  static void f() {
    throw new RuntimeException("Z f()");
  }
  static void g() {
    f();
  }
  public static void main(String[] args) {
```

```
    g();
  }
} ///:~
```

Widać już, że wyjątek `RuntimeException` (lub cokolwiek co po nim dziedziczy) jest tu specjalnym przypadkiem, ponieważ kompilator nie wymaga dla tych typów specyfikacji wyjątków. Wyjście jest zaś wypisywane do strumienia `System.err`:

```
Exception in thread "main" java.lang.RuntimeException: Z f()
        at NeverCaught.f(NeverCaught.java:7)
        at NeverCaught.g(NeverCaught.java:10)
        at NeverCaught.main(NeverCaught.java:13)
```

Odpowiedź brzmi zatem następująco — jeśli `RuntimeException` przedostanie się aż do `main()` i nie zostanie przechwycony, to przed wyjściem z programu wywoływana jest dla niego metoda `printStackTrace()`.

Należy pamiętać, że programując, można pominąć jedynie wyjątki typu `RuntimeException`, ponieważ obsługa wszystkich pozostałych jest wymuszona przez kompilator. Założono bowiem, że wyjątek `RuntimeException` ma reprezentować błędy programistyczne, takie jak:

1. Błąd, którego nie można przewidzieć. Na przykład przekazanie referencji `null` z kodu, którego programista nie kontroluje.
2. Błędy, które my, jako programiści, powinniśmy sami wykrywać we własnym kodzie (takie jak wyjątek `ArrayIndexOutOfBoundsException`, gdzie należałoby zwrócić uwagę na rozmiar tablicy). Powodem tych wyjątków często stają się wyjątki zaliczające się do poprzedniego punktu.

Widać tu, jaką olbrzymią zaletą jest w tym przypadku istnienie wyjątków: pomagają one w procesie testowania i usuwania błędów.

Warto zauważyć, że nie da się zaklasyfikować obsługi wyjątków w Javie jako narzędzia o jednoznacznym przeznaczeniu. Owszem, jest zaprojektowana do obsługi tych wrednych błędów wykonania, które występują z powodów leżących poza obszarem kontrolowanym przez nasz kod, ale jest również niezbędna do obsługi pewnych typów błędów programistycznych niewykrywalnych dla kompilatora.

Ćwiczenie 12. Zmodyfikuj plik *innerclasses/Sequence.java* tak, aby klasa reagowała na próbę wstawienia zbyt dużej liczby elementów wyrzuceniem odpowiedniego wyjątku (3).

Robienie porządków w finally

Często mamy pewien fragment kodu, który chcielibyśmy wykonać niezależnie od tego, czy w bloku `try` został zgłoszony wyjątek. Przeważnie odnosi się to do operacji innych niż odzyskiwanie pamięci (ponieważ tym zajmuje się odśmiecacz pamięci). Aby to osiągnąć, na końcu procedur wyjątków umieszcza się sekcję `finally`[5]. Zatem pełna składnia bloku obsługi wyjątków wygląda następująco:

[5] Obsługa wyjątków w C++ nie posiada sekcji `finally`, ponieważ opiera się na destruktorach, aby otrzymać przywracanie porządku w takim stylu.

```
try {
  // Obszar chroniony: Niebezpieczne działania,
  // które mogą zgłosić A, B lub C
} catch(A a1) {
  // Obsługa dla sytuacji A
} catch(B b1) {
  // Obsługa dla sytuacji B
} catch(C c1) {
  // Obsługa dla sytuacji C
} finally {
  // Czynności, które są wykonywane za każdym razem.
}
```

Aby zademonstrować, że sekcja `finally` jest zawsze wykonywana, spójrzmy na poniższy program:

```
//: exceptions/FinallyWorks.java
// Blok finally jest wykonywany _zawsze_

class ThreeException extends Exception {}

public class FinallyWorks {
  static int count = 0;
  public static void main(String[] args) {
    while(true) {
      try {
        // Postinkrementacja za pierwszym razem zwraca zero
        if(count++ == 0)
          throw new ThreeException();
        System.out.println("Brak wyjątku");
      } catch(ThreeException e) {
        System.out.println("Wyjątek ThreeException");
      } finally {
        System.out.println("W bloku finally");
        if(count == 2) break; // wyjście z pętli while
      }
    }
  }
} /* Output:
Wyjątek ThreeException
W bloku finally
Brak wyjątku
W bloku finally
*///:~
```

Wyniki generowane przez powyższy program pokazują, że niezależnie od tego, czy wyjątek zostanie zgłoszony czy nie, klauzula `finally` zawsze jest wykonywana.

Ten program pokazuje również, jak można sobie poradzić z problemem, o którym była mowa wcześniej, polegającym na tym, że wyjątki w Javie nie pozwalają na powrót do miejsca wyrzucenia wyjątku. Umieszczenie bloku `try` w pętli oznacza ustalenie warunków, które muszą zostać spełnione przed kontynuacją programu. Można również umieścić w pętli statyczny licznik lub jakieś inne rozwiązanie pozwalające pętli na wypróbowanie kilku różnych podejść przed poddaniem się. W ten sposób można osiągnąć większą niezawodność programu.

Do czego służy finally

W języku pozbawionym automatycznego zwalniania pamięci *oraz* automatycznego wywoływania destruktorów[6] sekcja `finally` jest istotna, ponieważ pozwala programiście zagwarantować zwolnienie pamięci niezależnie od tego, co może się stać w bloku try. Alc przecież Java posiada automatyczne zwalnianie pamięci, więc zwalnianie pamięci potencjalnie nigdy nie jest problemem. Nie posiada również żadnych destruktorów, które można by wywołać. Zatem kiedy może być w Javie konieczne użycie `finally`?

Sekcja `finally` jest konieczna, kiedy trzeba przywrócić do pierwotnego stanu coś *innego* niż pamięć. Jest to coś, co wymaga przywrócenia porządku, tak jak otwarty plik lub połączenie sieciowe, coś narysowanego na ekranie lub nawet przełącznik w świecie rzeczywistym, tak jak zostało to pokazane w poniższym przykładzie:

```
//: exceptions/Switch.java
import static net.mindview.util.Print.*;

public class Switch {
  private boolean state = false;
  public boolean read() { return state; }
  public void on() { state = true; print(this); }
  public void off() { state = false; print(this); }
  public String toString() { return state ? "wł" : "wył"; }
} ///:~

//: exceptions/OnOffException1.java
public class OnOffException1 extends Exception {} ///:~

//: exceptions/OnOffException2.java
public class OnOffException2 extends Exception {} ///:~

//: exceptions/OnOffSwitch.java
// Po co nam finally?

public class OnOffSwitch {
  private static Switch sw = new Switch();
  public static void f()
  throws OnOffException1,OnOffException2 {}
  public static void main(String[] args) {
    try {
      sw.on();
      // Kod, który potencjalnie wyrzuca wyjątki...
      f();
      sw.off();
    } catch(OnOffException1 e) {
      System.out.println("OnOffException1");
      sw.off();
    } catch(OnOffException2 e) {
      System.out.println("OnOffException2");
      sw.off();
```

[6] Destruktor to funkcja, która jest wywoływana zawsze, kiedy obiekt przestaje być używany. Zawsze wiadomo, kiedy i gdzie konstruktor jest wywoływany. C++ posiada automatyczne wywoływanie destruktorów, a C# (który o wiele bardziej przypomina Javę) dysponuje sposobem pozwalającym na automatyczne usunięcie obiektu.

```
    }
  }
} /* Output.
wł
wył
*///:~
```

Celem programu z tego przykładu jest zapewnienie, że przełącznik będzie wyłączony po zakończeniu `main()`, zatem wywołanie `sw.off()` jest wstawione na końcu bloku `try` i na końcu każdej procedury obsługi wyjątku. Ale możliwe jest, że zostanie wyrzucony wyjątek, który nie zostanie tu przechwycony, więc wywołanie `sw.off()` zostanie pominięte. Używając `finally`, można umieścić w jednym miejscu kod, który wykona porządki po całym bloku `try`.

```
//: exceptions/WithFinally.java
// Blok finally gwarantuje przeprowadzenie porządków.

public class WithFinally {
  static Switch sw = new Switch();
  public static void main(String[] args) {
    try {
      sw.on();
      // Kod potencjalnie wyrzucający wyjątki...
      OnOffSwitch.f();
    } catch(OnOffException1 e) {
      System.out.println("OnOffException1");
    } catch(OnOffException2 e) {
      System.out.println("OnOffException2");
    } finally {
      sw.off();
    }
  }
} /* Output:
wł
wył
*///:~
```

Tutaj wywołanie `sw.off()` zostało przeniesione w jedno miejsce, w którym jego wykonanie jest zapewnione niezależnie od tego, co się stanie.

Nawet w przypadku, kiedy wyjątek nie zostanie przechwycony w aktualnym zestawie bloków `catch`, sekcja `finally` zostanie wykonana, zanim mechanizm obsługi wyjątków zacznie kontynuować poszukiwanie procedury obsługi wyjątku na wyższym poziomie:

```
//: exceptions/AlwaysFinally.java
// Blok finally jest wykonywany „zawsze".
import static net.mindview.util.Print.*;

class FourException extends Exception {}

public class AlwaysFinally {
  public static void main(String[] args) {
    print("Wejscie do pierwszego bloku try");
    try {
      print("Wejście do drugiego bloku try");
      try {
        throw new FourException();
```

```
      } finally {
        print("Blok finally w drugim bloku try");
      }
    } catch(FourException e) {
      System.out.println(
        "Przechwycono FourException w pierwszym bloku try");
    } finally {
      System.out.println("Blok finally w pierwszym bloku try");
    }
  }
} /* Output:
Wejście do pierwszego bloku try
Wejście do drugiego bloku try
Blok finally w drugim bloku try
Przechwycono FourException w pierwszym bloku try
Blok finally w pierwszym bloku try
*///:~
```

Instrukcja `finally` zostanie wykonana również w sytuacjach związanych z instrukcją `break` i `continue`. Wraz z etykietowanym `break` i etykietowanym `continue` instrukcja `finally` eliminuje potrzebę używania w Javie instrukcji `goto`.

Ćwiczenie 13. Zmodyfikuj ćwiczenie 9. przez dodanie sekcji `finally`. Sprawdź, czy sekcja `finally` jest wykonywana nawet wtedy, gdy zgłaszany jest wyjątek `NullPointerException` (2).

Ćwiczenie 14. Pokaż, że program *OnOffSwitch.java* może przestać działać przez wyrzucenie wyjątku `RuntimeException` wewnątrz bloku `try` (2).

Ćwiczenie 15. Pokaż, że program *WithFinally.java* będzie działać po zgłoszeniu wyjątku `RuntimeException` wewnątrz bloku `try` (2).

Współdziałanie finally z return

Skoro kod bloku `finally` jest wykonywany zawsze, to w obrębie metody można umieszczać wiele instrukcji powrotu (`return`), zachowując pewność, że za każdym razem powrót zostanie poprzedzony wymaganymi porządkami:

```
//: exceptions/MultipleReturns.java
import static net.mindview.util.Print.*;

public class MultipleReturns {
  public static void f(int i) {
    print("Inicjalizacja wymagająca porządkowania");
    try {
      print("Punkt 1");
      if(i == 1) return;
      print("Punkt 2");
      if(i == 2) return;
      print("Punkt 3");
      if(i == 3) return;
      print("Koniec");
      return;
    } finally {
      print("Porządki");
    }
```

```
  }
  public static void main(String[] args) {
    for(int i = 1; i <= 4; i++)
      f(i);
  }
} /* Output:
Inicjalizacja wymagająca porządkowania
Punkt 1
Porządki
Inicjalizacja wymagająca porządkowania
Punkt 1
Punkt 2
Porządki
Inicjalizacja wymagająca porządkowania
Punkt 1
Punkt 2
Punkt 3
Porządki
Inicjalizacja wymagająca porządkowania
Punkt 1
Punkt 2
Punkt 3
Koniec
Porządki
*///:~
```

Na wyjściu widać, że niezależnie od punktu powrotu z metody zawsze dochodzi do wykonania kodu z `finally`.

Ćwiczenie 16. Zmień przykład *reusing/CADSystem.java*, demonstrując w nim powracanie ze środka bloku `try-finally` przy każdorazowym zachowaniu wymaganych operacji porządkujących (2).

Ćwiczenie 17. Zmień przykład *polymorphism/Frog.java* tak, aby gwarantował odpowiednie porządkowanie za pomocą konstrukcji `try-finally`; pokaż, że porządkowanie odbywa się nawet, jeśli wymusimy powrót ze środka `try-finally` (3).

Pułapka: zagubiony wyjątek

Niestety, implementacja wyjątków w Javie posiada słaby punkt. Mimo że wyjątki sygnalizują sytuacje kryzysowe w programie i nigdy nie powinny być ignorowane, możliwe jest jednak zgubienie wyjątku. Zdarza się to przy pewnym szczególnym ustawieniu, kiedy używa się sekcji `finally`:

```
//: exceptions/LostMessage.java
// Jak można zgubić wyjątek.

class VeryImportantException extends Exception {
  public String toString() {
    return "Bardzo ważny wyjątek!";
  }
}

class HoHumException extends Exception {
  public String toString() {
    return "Banalny wyjątek";
```

```
  }
}

public class LostMessage {
  void f() throws VeryImportantException {
    throw new VeryImportantException();
  }
  void dispose() throws HoHumException {
    throw new HoHumException();
  }
  public static void main(String[] args) {
    try {
      LostMessage lm = new LostMessage();
      try {
        lm.f();
      } finally {
        lm.dispose();
      }
    } catch(Exception e) {
      System.out.println(e);
    }
  }
} /* Output:
Banalny wyjątek
*///:~
```

Widać tu, że zgubiono wszelki ślad po wyjątku `VeryImportantException`, który został po prostu zastąpiony w sekcji `finally` wyjątkiem `HoHumException`. Jest to dość poważna pułapka, gdyż oznacza, że wyjątek może być całkowicie zgubiony, nawet w sytuacjach bardziej subtelnych i trudniejszych do wykrycia niż w powyższym przykładzie. Dla porównania C++ traktuje sytuację, w której drugi wyjątek jest wyrzucany, zanim pierwszy zostanie obsłużony, jako skrajnie niebezpieczny błąd programistyczny. Być może przyszłe wersje Javy rozwiążą ten problem (z drugiej strony metody zgłaszające wyjątek, takie jak `dispose()`, opakowuje się przeważnie blokiem `try-catch`).

Wyjątek można zresztą zgubić jeszcze prościej: wystarczy wykonać instrukcję `return` w bloku `finally`:

```
//: exceptions/ExceptionSilencer.java

public class ExceptionSilencer {
  public static void main(String[] args) {
    try {
      throw new RuntimeException();
    } finally {
      // Użycie 'return' w bloku finally
      // tłumi wszelkie wyrzucone wyjątki
      return;
    }
  }
} ///:~
```

Ćwiczenie 18. Dodaj drugi poziom wyjątków do programu *LostMessage.java* tak, aby wyjątek `HoHumException` był sam zastępowany przez inny wyjątek (3).

Ćwiczenie 19. Usuń problem z pliku *LostMessage.java* przez ochronę wywołania w bloku `finally` (2).

Ograniczenia wyjątków

W metodzie przeciążonej można zgłaszać jedynie te wyjątki, które zostały podane w specyfikacji jej wersji z klasy bazowej. Jest to użyteczne ograniczenie, ponieważ oznacza, że kod, który działa dla klasy bazowej, będzie automatycznie działał z każdym obiektem dziedziczącym z klasy bazowej (oczywiście jest to podstawowe założenie programowania zorientowanego obiektowo), włączając w to prawidłową obsługę wyjątków.

Poniższy przykład pokazuje rodzaje ograniczeń narzucone (przez kompilator) na obsługę wyjątków:

```
//: exceptions/StormyInning.java
// Metody przesłonięte mogą wyrzucać jedynie te wyjątki,
// które zostały określone w wersjach z klasy bazowej,
// ewentualnie pochodne wyjątków tam określonych.

class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
  public Inning() throws BaseballException {}
  public void event() throws BaseballException {
    // Nie musi niczego wyrzucać
  }
  public abstract void atBat() throws Strike, Foul;
  public void walk() {} // Nie wyrzuca sprawdzanych wyjątków
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  public void event() throws RainedOut;
  public void rainHard() throws RainedOut;
}

public class StormyInning extends Inning implements Storm {
  // Można dodawać nowe wyjątki dla konstruktorów, ale trzeba
  // poradzić sobie z wyjątkami konstruktora klasy bazowej:
  public StormyInning()
    throws RainedOut, BaseballException {}
  public StormyInning(String s)
    throws Foul, BaseballException {}
  // Zwykłe metody muszą trzymać się specyfikacji wyjątków
  // według wersji z klasy bazowej:
//! void walk() throws PopFoul {} // Błąd kompilacji
  // Interfejs NIE MOŻE dodawać wyjątków do istniejących
  // metod klasy bazowej:
//! public void event() throws RainedOut {}
  // Jeśli metoda nie istnieje w klasie bazowej,
  // można jej nadać dowolną specyfikację wyjątków
  public void rainHard() throws RainedOut {}
  // Można zaniechać wyrzucania jakichkolwiek wyjątków
```

```
    // pomimo ich obecności w klasie bazowej:
    public void event() {}
    // Metody przesłonięte mogą wyrzucać wyjątki pochodne
    public void atBat() throws PopFoul {}
    public static void main(String[] args) {
      try {
        StormyInning si = new StormyInning();
        si.atBat();
      } catch(PopFoul e) {
        System.out.println("Pop foul");
      } catch(RainedOut e) {
        System.out.println("Rained out");
      } catch(BaseballException e) {
        System.out.println("Ogólny wyjątek rozgrywki");
      }
      // Wyjątek Strike nie jest zgłaszany w wersji przeciążonej.
      try {
        // Co się stanie przy rzutowaniu w górę?
        Inning i = new StormyInning();
        i.atBat();
        // Trzeba przechwycić wyjątki z wersji metody
        // z klasy bazowej
      } catch(Strike e) {
        System.out.println("Strike");
      } catch(Foul e) {
        System.out.println("Foul");
      } catch(RainedOut e) {
        System.out.println("Rained out");
      } catch(BaseballException e) {
        System.out.println("Ogólny wyjątek rozgrywki");
      }
    }
} ///:~
```

W klasie `Inning` zarówno konstruktor, jak i metoda `event()` podają, że będą zgłaszać wyjątki — mimo iż w rzeczywistości nigdy tego nie robią. Jest to możliwe, ponieważ wymusza na użytkowniku przechwytywanie wyjątku, który może zostać dodany dopiero w przeciążonej wersji metody `event()`. To samo odnosi się do metod typu `abstract`, takich jak `atBat()`.

Interfejs `Storm` jest ciekawy, gdyż zawiera jedną metodę (`event()`), która jest definiowana w `Inning`, oraz jedną, która nie jest tam definiowana. Obie metody wyrzucają nowy rodzaj wyjątku — `RainedOut`. Kiedy klasa `StormyInning` rozszerza klasę `Inning` i implementuje interfejs `Storm`, to metoda `event()` w `Storm()` *nie może* zmienić specyfikacji wyjątków metody z `Inning`. I to również ma sens, ponieważ gdyby tak nie było, nigdy nie byłoby wiadomo, czy jeśli ograniczymy się do obsługi wyjątków z klasy bazowej, to zostaną przechwycone wszystkie wyjątki. Oczywiście jeśli metoda zdefiniowana w interfejsie nie istnieje w klasie bazowej, tak jak `rainHard()`, to nie ma żadnego problemu, jeśli zgłasza ona wyjątki.

Ograniczenia wyjątków nie stosują się do konstruktorów. W klasie `StormyInning` widać, że konstruktor może zgłaszać co zechce, niezależnie od tego, co zgłasza konstruktor klasy bazowej. Jednak, ponieważ konstruktor klasy bazowej musi zostać wywołany w taki czy inny sposób (tutaj domyślny konstruktor był wywoływany automatycznie), konstruktor klasy pochodnej musi zadeklarować wszystkie wyjątki konstruktora klasy bazowej.

Konstruktor klasy pochodnej nie może przechwytywać wyjątków zgłaszanych przez konstruktor klasy bazowej.

Powodem, dla którego metoda `StormyInning.walk()` nie będzie skompilowana, jest to, że zgłasza ona wyjątek, podczas gdy `Inning.walk()` go nie zgłasza. Gdyby było to dozwolone, można by pisać kod, który wywołuje `Inning.walk()` i nie musi obsługiwać żadnych wyjątków. Natomiast jeśli później zostanie do niego podstawiony obiekt klasy dziedziczącej po `Inning`, który zgłosi wyjątek, to pierwotny kod nie będzie mógł go obsłużyć. Przez wymuszenie na metodach klas pochodnych zgodności ze specyfikacją wyjątków w metodach klas bazowych zachowana jest możliwość podstawienia innego obiektu.

Przesłonięta metoda `event()` pokazuje, że metoda w wersji z klasy pochodnej może wcale nie zgłaszać wyjątków, nawet jeśli robi to klasa bazowa. I znów jest to założenie poprawne, ponieważ nie wywołuje żadnych błędów w istniejącym kodzie zakładając, że wersja metody z klasy bazowej zgłasza wyjątki. Podobne rozumowanie odnosi się do metody `atBat()`, która zgłasza `PopFoul` — wyjątek, który jest odziedziczony po wyjątku `Foul` zgłaszanym przez bazową wersję metody `atBat()`. Jeśli więc ktoś napisze kod, który działa z klasą `Inning` i wywołuje `atBat()`, to będzie musiał przechwycić wyjątek `Foul`. Ponieważ `PopFoul` dziedziczy po `Foul`, procedura obsługująca wyjątek `Foul` przechwyci również wyjątek `PopFoul`.

Ostatnim interesującym miejscem jest metoda `main()`. Można zauważyć, że jeśli pracujemy z obiektem typu `StormyInning`, to kompilator zmusza nas do przechwycenia tylko tych wyjątków, które są specyficzne dla tej klasy. Natomiast jeśli zrzutujemy ten obiekt na klasę bazową, to kompilator (prawidłowo) zmusza nas do przechwycenia wyjątków klasy bazowej. Wszystkie te ograniczenia prowadzą do bardziej niezawodnej obsługi wyjątków[7].

Mimo że specyfikacja wyjątków nie jest częścią definicji metody, która jest określona wyłącznie przez nazwę metody i typy jej parametrów, i to mimo iż specyfikacje wyjątków są wymuszane przez kompilator w trakcie dziedziczenia. Zatem nie można przeciążać metod, posługując się specyfikacją wyjątków. Dodatkowo to, że specyfikacja wyjątków istnieje w wersji metody w klasie bazowej, wcale nie oznacza, że musi ona istnieć w wersji odziedziczonej tej metody. Jest to całkowite przeciwieństwo zasady dziedziczenia, gdzie metoda z klasy bazowej musi istnieć również w klasie pochodnej. Innymi słowy: „interfejs specyfikacji wyjątków" dla konkretnej metody może zostać zawężony w trakcie dziedziczenia i przesłonięcia, ale nie może się rozszerzać — jest to dokładna odwrotność zasady zmieniania w trakcie dziedziczenia interfejsu klasy.

Ćwiczenie 20. Zmodyfikuj program *StormyInning.java*, dodając wyjątek typu `UmpireArgument` (decyzja sędziego) i metody zgłaszające ten wyjątek. Przetestuj zmodyfikowaną hierarchię (3).

[7] ISO C++ dodaje podobne ograniczenia, które wymagają, by wyjątki metody pochodnej były takie same lub odziedziczone po wyjątkach zgłaszanych przez metodę klasy bazowej. Jest to jedyny przypadek, kiedy C++ może sprawdzić specyfikację wyjątków w momencie kompilacji.

Konstruktory

Zawsze należy zadawać sobie pytanie: „czy w razie wyjątku wszystko zostanie odpowiednio uporządkowane?". Przez większość czasu jesteśmy względnie bezpieczni, ale problem ten może się pojawić przy konstruktorach. Konstruktor ustawia obiekt w bezpiecznym stanie początkowym, ale może wykonać jakąś operację — jak otwarcie pliku — która nie zostanie cofnięta, dopóki użytkownik nie zakończy używania obiektu i nie wywoła specjalnej metody sprzątającej. Jeśli wyjątek zostanie zgłoszony wewnątrz konstruktora, to kod sprzątający może nie zadziałać prawidłowo. Oznacza to, że pisząc konstruktor, trzeba zachować szczególną ostrożność.

Wydawać by się mogło, że rozwiązaniem jest `finally`. Nie jest to jednak takie proste, ponieważ kod sprzątający w `finally` jest wykonywany *za każdym razem*, nawet w sytuacjach, w których tego nie chcemy. Tymczasem jeśli konstruktor wykona się tylko częściowo, może nie zdążyć utworzyć czy zainicjalizować obiektów, które miałyby być porządkowane w bloku `finally`.

W poniższym przykładzie tworzona jest klasa `InputFile`, która otwiera plik i pozwala czytać z niego po jednym wierszu (skonwertowanym na `String`). Używa ona klasy `FileReader` i `BufferedReader` ze standardowej biblioteki wejścia-wyjścia Javy, które zostaną omówione w rozdziale „Wejście-wyjście". Klasy te są na tyle proste, że zrozumienie podstaw ich stosowania nie powinno przysporzyć nikomu większych trudności:

```
//: exceptions/InputFile.java
// Uwaga na wyjątki w konstruktorach.
import java.io.*;

public class InputFile {
  private BufferedReader in;
  public InputFile(String fname) throws Exception {
    try {
      in = new BufferedReader(new FileReader(fname));
      // Reszta kodu, który mógłby spowodować wyjątki
    } catch(FileNotFoundException e) {
      System.out.println("Nie można otworzyć pliku " + fname);
      // Plik nie był otwarty, więc go nie zamykamy
      throw e;
    } catch(Exception e) {
      // Wszystkie pozostałe wyjątki muszą zadbać o zamknięcie pliku
      try {
        in.close();
      } catch(IOException e2) {
        System.out.println("wywołanie in.close() nieskuteczne");
      }
      throw e; // Ponowne wyrzucenie wyjątku
    } finally {
      // Nie zamykać pliku tutaj!!!
    }
  }
  public String getLine() {
    String s;
    try {
      s = in.readLine();
```

```
      } catch(IOException e) {
        throw new RuntimeException("wywołanie readLine() nieskuteczne");
      }
      return s;
    }
    public void dispose() {
      try {
        in.close();
        System.out.println("wywołanie dispose() skuteczne");
      } catch(IOException e2) {
        throw new RuntimeException("wywołanie in.close() nieskuteczne");
      }
    }
  } ///:~
```

Konstruktor klasy `InputFile` przyjmuje pojedynczy parametr typu `String`, będący nazwą pliku, który chcemy otworzyć. Wewnątrz bloku `try` przy użyciu tej nazwy pliku tworzony jest obiekt typu `FileReader`. Obiekt `FileReader` nie jest szczególnie przydatny, o ile nie użyje się go do stworzenia obiektu typu `BufferedReader`. Jedną z zalet tworzonej przez nas klasy `InputFile` jest to, że łączy ona te dwie czynności.

Jeśli wykonanie konstruktora `FileReader` nie powiedzie się, zgłasza on wyjątek `FileNotFoundException`. Jest to jedyny przypadek, w którym nie chcemy zamykać pliku, gdyż właśnie nie udało się go otworzyć. Wszystkie *pozostałe* bloki `catch` muszą zamknąć plik, ponieważ *był* on otwarty w momencie wejścia do tych bloków (oczywiście sytuacja robi się bardziej skomplikowana, jeśli więcej niż jedna metoda może zgłosić wyjątek `FileNotFoundException`. W takim przypadku można próbować rozbić całość na kilka bloków `try`). Metoda `close()` może również zgłosić wyjątek, a więc jest umieszczona w bloku `try-catch`, mimo że jest już w bloku innej sekcji `catch` — dla kompilatora Javy jest to tylko jeszcze jedna para nawiasów klamrowych. Po wykonaniu lokalnych operacji wyjątek jest wyrzucany ponownie, co jest prawidłowe, ponieważ wykonanie konstruktora nie powiodło się i nie chcemy, aby metoda, która go wywołała, zakładała, że obiekt został stworzony właściwie i jest poprawny.

W tym przykładzie sekcja `finally` zdecydowanie *nie jest* odpowiednim miejscem na zamknięcie pliku — spowodowałoby to zamknięcie go przy każdym wykonaniu konstruktora. Takie zachowanie nie byłoby prawidłowe, ponieważ chcemy, aby plik był otwarty tak długo, jak długo używany jest obiekt `InputFile`.

Metoda `getLine()` zwraca łańcuch zawierający kolejny wiersz pliku. Wywołuje ona metodę `readLine()`, która może zgłosić wyjątek, ale wyjątek jest przechwytywany, więc `getLine()` nie zgłasza żadnych wyjątków. Jedną z kwestii do rozstrzygnięcia przy projektowaniu jest to, czy obsłużyć wyjątek całkowicie na danym poziomie, czy obsłużyć go częściowo i przekazać ten sam (lub inny) wyjątek dalej, czy też po prostu go przepuścić. Przepuszczenie, kiedy jest uzasadnione, może oczywiście uprościć programowanie. W tej sytuacji metoda `getLine()` *konwertuje* wyjątek do typu `RuntimeException`, aby poinformować o błędzie programisty.

Kiedy obiekt `InputFile` przestanie być potrzebny, należy wywołać metodę `dispose()`. Spowoduje ona zwolnienie zasobów systemowych (takich jak uchwyty plików) używanych w obiektach `BufferedReader` i (lub) `FileReader`. Nie chcemy tego robić, zanim nie

zakończymy korzystania z obiektu `InputFile`. Odpowiednie może się wydawać umieszczenie takich czynności w metodzie `finalize()`, ale — jak wspomniałem w rozdziale „Inicjalizacja i sprzątanie" — nie zawsze można zagwarantować, że `finalize()` zostanie wykonana (a nawet jeśli *można* mieć pewność, że zostanie ona wywołana, nigdy nie wiadomo, *kiedy* to nastąpi). Jest to jedna z wad Javy: żadne porządki — poza zwalnianiem pamięci — nie są wykonywane automatycznie, więc trzeba poinstruować programistę korzystającego z naszych klas, że jest za to odpowiedzialny.

Najbezpieczniejszym sposobem użycia klasy, która może podczas konstrukcji wyrzucić wyjątek i która wymaga operacji porządkujących, jest użycie zagnieżdżonych bloków `try`:

```
//: exceptions/Cleanup.java
// Zapewnianie właściwego uporządkowania stanu zasobów

public class Cleanup {
  public static void main(String[] args) {
    try {
      InputFile in = new InputFile("Cleanup.java");
      try {
        String s;
        int i = 1;
        while((s = in.getLine()) != null)
          ; // Tu przetwarzanie wiersz po wierszu...
      } catch(Exception e) {
        System.out.println("Caught Exception in main");
        e.printStackTrace(System.out);
      } finally {
        in.dispose();
      }
    } catch(Exception e) {
      System.out.println("Błąd konstrukcji obiektu InputFile");
    }
  }
} /* Output:
wywołanie dispose() skuteczne
*///:~
```

Przyjrzyj się dobrze logice tego kodu: operacja konstrukcji obiektu `InputFile` jest umieszczona w jej własnym bloku `try`. Jeśli owa konstrukcja zawiedzie, dojdzie do wywołania zewnętrznej klauzuli `catch` i pominięcia wywołania `dispose()`. Jeśli jednak konstrukcja się powiedzie, wtedy interesuje nas zapewnienie odpowiedniego uporządkowania obiektu, więc zaraz po konstrukcji rozpoczynamy następny blok `try`. Blok `finally`, realizujący operacje porządkowe, jest skojarzony właśnie z tym wewnętrznym `try`; w ten sposób klauzula `finally` jest pomijana w przypadku błędu konstrukcji obiektu, ale wykonywana *zawsze*, jeśli obiekt zostanie pomyślnie utworzony.

Taki idiom porządkowania można wykorzystać również wtedy, kiedy konstruktor nie wyrzuca żadnych wyjątków. Podstawowa reguła brzmi prosto: zaraz po utworzeniu obiektu wymagającego porządkowania zacznij blok `try-finally`:

```
//: exceptions/CleanupIdiom.java
// Każdy obiekt musi mieć własny blok try-finally

class NeedsCleanup { // Konstrukcja zawsze udana
  private static long counter = 1;
```

```
  private final long id = counter++;
  public void dispose() {
    System.out.println("NeedsCleanup " + id + " uporządkowany");
  }
}

class ConstructionException extends Exception {}

class NeedsCleanup2 extends NeedsCleanup {
  // Konstrukcja może zawodzić:
  public NeedsCleanup2() throws ConstructionException {}
}

public class CleanupIdiom {
  public static void main(String[] args) {
    // Sekcja 1:
    NeedsCleanup nc1 = new NeedsCleanup();
    try {
      // ...
    } finally {
      nc1.dispose();
    }

    // Sekcja 2:
    // Jeśli konstrukcja jet zawsze udana, można zgrupować oiekty:
    NeedsCleanup nc2 = new NeedsCleanup();
    NeedsCleanup nc3 = new NeedsCleanup();
    try {
      // ...
    } finally {
      nc3.dispose(); // Kolejność odwrotna do kolejności konstrukcji
      nc2.dispose();
    }

    // Sekcja 3:
    // Jeśli konstrukcja może zawieść,
    // trzeba chronić każdy obiekt z osobna
    try {
      NeedsCleanup2 nc4 = new NeedsCleanup2();
      try {
        NeedsCleanup2 nc5 = new NeedsCleanup2();
        try {
          // ...
        } finally {
          nc5.dispose();
        }
      } catch(ConstructionException e) { // Konstruktor nc5
        System.out.println(e);
      } finally {
        nc4.dispose();
      }
    } catch(ConstructionException e) { // Konstruktor nc4
      System.out.println(e);
    }
  }
} /* Output:
NeedsCleanup 1 uporządkowany
```

```
NeedsCleanup 3 uporządkowany
NeedsCleanup 2 uporządkowany
NeedsCleanup 5 uporządkowany
NeedsCleanup 4 uporządkowany
*///:~
```

Sekcja 1. w metodzie `main()` jest całkiem prosta: konstrukcję obiektu wymagającego specjalnego trybu usuwania uzupełniamy blokiem `try-finally`. Jeśli konstrukcja nie może zawieść, nie trzeba stosować klauzuli `catch`. W sekcji 2. widać, że obiekty z konstruktorami, które nie wyrzucają wyjątków, można grupować we wspólnym bloku konstrukcji i porządkowania.

Sekcja 3. ilustruje sposób radzenia sobie z obiektami, których konstrukcja może powodować zgłoszenie wyjątków. W takim układzie sprawa się nieco komplikuje, bo każdą konstrukcję trzeba ujmować w bloku `try-catch`, a potem natychmiast uzupełniać blokiem `try-finally`.

Zagmatwanie strategii obsługi wyjątków w omawianym przypadku to mocny argument za tworzeniem niezawodnych (w kontekście wyjątków) konstruktorów — niestety, nie zawsze jest to możliwe.

Ćwiczenie 21. Pokaż, że konstruktor klasy pochodnej nie może przechwytywać wyjątków zgłaszanych przez konstruktor klasy bazowej (2).

Ćwiczenie 22. Utwórz klasę o nazwie `FailingConstructor` z konstruktorem, który może zawieść i wyrzucić wyjątek w połowie procesu konstrukcji. W metodzie `main()` napisz kod, który będzie prawidłowo chronił program przed tego rodzaju błędem (2).

Ćwiczenie 23. Dodaj do poprzedniego ćwiczenia klasę z metodą `dispose()`. Zmodyfikuj konstruktor `FailingConstructor` tak, aby konstruktor tworzył jeden z obiektów owej klasy w roli składowych obiektu konstruowanego, a następnie wyrzucał wyjątek, po czym tworzył drugi składowy obiekt klasy z metodą `dispose()`. Napisz kod chroniący przed błędem; sprawdź w `main()`, czy udało Ci się zabezpieczyć przed wszystkimi możliwymi błędami (4).

Ćwiczenie 24. Dodaj do `FailingConstructor` metodę `dispose()` i napisz kod, który prawidłowo korzysta z takiej klasy (3).

Dopasowywanie wyjątków

Gdy wyjątek jest zgłaszany, system obsługi wyjątków szuka „najbliższej" procedury obsługi w takiej kolejności, w jakiej są one napisane. Kiedy znajdzie pasującą, to wyjątek jest uznawany za obsłużony i nie następuje żadne dalsze przeszukiwanie.

Wyszukiwanie wyjątku nie wymaga dokładnego dopasowania wyjątku i procedury obsługi. Obiekt klasy pochodnej będzie pasował do procedury obsługującej wyjątek klasy bazową, tak jak w tym przykładzie:

```
//: exceptions/Human.java
// Przechwytywanie hierarchii wyjątków.

class Annoyance extends Exception {}
class Sneeze extends Annoyance {}

public class Human {
  public static void main(String[] args) {
    // Przechwytywanie wyjątku dokładnego typu:
    try {
      throw new Sneeze();
    } catch(Sneeze s) {
      System.out.println("Przechwycono Sneeze");
    } catch(Annoyance a) {
      System.out.println("Przechwycono Annoyance");
    }
    // Przechwytywanie typu bazowego
    try {
      throw new Sneeze();
    } catch(Annoyance a) {
      System.out.println("Przechwycono Annoyance");
    }
  }
} /* Output:
Przechwycono Sneeze
Przechwycono Annoyance
*///:~
```

Wyjątek Sneeze zostanie przechwycony przez pierwszy blok catch, do którego będzie pasował — czyli oczywiście przez pierwszy. Jednak jeśli pierwszy blok catch zostanie usunięty i pozostanie jedynie klauzula catch dla typu Annoyance, to kod będzie nadal poprawny, ponieważ przechwytuje klasę bazową dla Sneeze. Innymi słowy: blok catch(Annoyance a) przechwyci wyjątek Annoyance lub każdą klasę *dziedziczącą po nim*. Jest to przydatne, ponieważ jeśli zdecydujemy się na dodanie do metody kolejnych dziedziczonych wyjątków, to nie będzie potrzeby zmieniania kodu wywołującego tę metodę, o ile przechwytuje on już wyjątek bazowy.

Próba „przesłonięcia" obsługi wyjątków klas pochodnych przez umieszczenie najpierw bloku catch dla klasy bazowej, jak poniżej:

```
try {
  throw new Sneeze();
} catch(Annoyance a) {
  // ...
} catch(Sneeze s) {
  // ...
}
```

spowoduje pojawienie się komunikatu o błędzie w trakcie kompilacji, ponieważ kompilator widzi, że procedura obsługi wyjątku Sneeze nie zostanie nigdy wykonana.

Ćwiczenie 25. Stwórz trójpoziomową hierarchię wyjątków. Następnie stwórz klasę bazową A z metodą, która zgłasza wyjątek będący podstawą hierarchii. Odziedzicz B z A i przeciąż tę metodę tak, żeby zgłaszała wyjątek na drugim poziomie hierarchii. Powtórz to, dziedzicząc klasę C z B. W main() utwórz obiekt klasy C i zrzutuj go do A, a następnie wywołaj jego metodę (2).

Rozwiązania alternatywne

System obsługi wyjątków jest awaryjnym wyjściem pozwalającym programowi na porzucenie normalnej ścieżki wykonywania instrukcji. To wyjście awaryjne jest wykorzystywane, gdy zajdzie „sytuacja wyjątkowa", w której zwyczajny sposób działania jest niemożliwy lub niepożądany. Wyjątki odpowiadają zatem warunkom, z którymi metoda nie jest sobie w stanie poradzić. Systemy obsługi wyjątków zostały stworzone, ponieważ rozwiązanie polegające na obsłudze wszystkich możliwych błędów, które mogły być generowane przez wszystkie używane funkcje, było zbyt uciążliwe, a programiści po prostu go nie stosowali, w rezultacie ignorując zgłaszane błędy. Warto zauważyć, że podstawowym czynnikiem branym pod uwagę podczas tworzenia mechanizmów obsługi wyjątków, była wygoda programisty.

Jedna z podstawowych zasad obsługi wyjątków zaleca, by: „nie przechwytywać wyjątku, jeśli nie wiadomo, co z nim zrobić". W rzeczywistości jednym z ważnych *celów* wprowadzenia obsługi błędów była chęć oddzielenia kodu obsługującego błąd od kodu, w którym błąd ten powstał. Dzięki takiemu rozwiązaniu w danym fragmencie kodu można skoncentrować się na realizacji zamierzonych czynności, a w innej, niezależnej części kodu na sposobach rozwiązania ewentualnych problemów. Dzięki temu najważniejsze części programu nie zawierają fragmentów związanych z obsługą błędów, przez co są łatwiejsze do zrozumienia i utrzymania. Obsługa wyjątków skupia też kod obsługi błędów, eliminując rozproszenie kodu obsługi po całym programie.

Scenariusz ten komplikują nieco wyjątki sprawdzane, wymagające dodania odpowiednich klauzul `catch` w miejscach, w których należy być przygotowanym do obsługi wyjątków. W ten sposób powstaje problem pojawiania się zagrożenia w razie zignorowania wyjątku:

```
try {
  // ... zrób coś przydatnego
} catch (WyjątekObowiązkowoObsługiwany e) {}  // "połykamy" wyjątek
```

Programiści (w czasie tworzenia pierwszego wydania tej książki, dotyczyło to także mnie) wykorzystaliby zapewne najprostsze rozwiązanie i zignorowali („połknęli") wyjątek. Metoda ta jest często stosowana nieświadomie, lecz jeśli się ją już zastosuje, kompilator nie będzie zgłaszać żadnych błędów. Oznacza to, że wyjątek może zostać stracony, chyba że programista będzie pamiętał, by zmodyfikować odpowiedni fragment kodu. W takim przypadku wyjątek jest zgłaszany, lecz w efekcie „połknięcia" całkowicie znika. Kompilator zmusza nas do natychmiastowego tworzenia procedur obsługi wyjątków, dlatego też powyższe rozwiązanie zdaje się być najprostsze, choć zapewne jest jednocześnie najgorszym z możliwych.

Przerażony tym, co zrobiłem w pierwszym wydaniu książki, w drugim „rozwiązałem" problem, wyświetlając w procedurach obsługi błędów zrzuty stosu (rozwiązanie to wciąż jest wykorzystywane w wielu przykładach przedstawionych w tym rozdziale). Choć prezentacja informacji o stosie jest przydatna, nadal oznacza, że w danym miejscu kodu nie wiadomo, co należy zrobić z wyjątkiem. W tej części rozdziału przyjrzymy się pewnym zagadnieniom oraz problemom, jakich przysparzają wyjątki sprawdzane, jak również możliwościom obsługi takich wyjątków.

Zagadnienie wydaje się proste, jednak w praktyce jest skomplikowane, a co więcej, ma związek z ulotnością. Są osoby wiernie trzymające się jednej strony barykady, uważające, że poprawna odpowiedź (ich odpowiedź) jest najzupełniej oczywista. Sądzę, że przyczyną takiej postawy mogą być korzyści zauważalne przy przechodzeniu z języków słabo typowanych, takich jak wersje C powstałe przed pojawieniem się standardu ANSI C, do języków kontrolujących typy bardzo rygorystycznie i statycznie (czyli już w czasie kompilacji), takich jak C++ czy Java. W momencie zmieniania używanego języka korzyści te są tak wyraźnie i znaczące, że rygorystyczna, statyczna kontrola typów może się zdawać najlepszym rozwiązaniem większości problemów. Chciałbym odnieść się nieco do własnej ewolucji, która doprowadziła mnie do zakwestionowania *bezwzględnych* zalet rygorystycznego, statycznego sprawdzania typów. Bez wątpienia w większości przypadków jest ono bardzo przydatne, istnieje jednak ulotna granica, po której przekroczeniu staje się ono przeszkodą. (Moje ulubione powiedzenie to: „Wszystkie modele są złe. Niektóre są przydatne.")

Historia

Mechanizmy obsługi wyjątków pojawiły się początkowo w takich systemach jak PL/1 i Mesa, następnie w językach CLU, Smalltalk, Modula-3, Ada, Eiffel, C++, Python, Java, a w końcu w językach wzorowanych na Javie — Ruby i C#. Rozwiązania wykorzystane w Javie są wzorowane na języku C++, z wyjątkiem tych sytuacji, w których twórcy Javy doszli do wniosku, że mechanizmy C++ mogą stwarzać problemy.

Obsługa wyjątków została dodana do języka C++ na stosunkowo późnym etapie standaryzacji. Miała ona na celu udostępnienie szkieletu obsługi wyjątków i rozwiązywania problemów, który byłby powszechniej wykorzystywany przez programistów, i była promowana przez autora języka — Bjarne'a Stroustrupa. Model obsługi wyjątków przyjęty w C++ pochodził przede wszystkim z języka CLU. Niemniej jednak w tym czasie istniały także inne języki dysponujące analogicznymi rozwiązaniami. Należały do nich: Ada, Smalltalk (oba te języki obsługiwały wyjątki, lecz nie dysponowały możliwością ich specyfikacji) oraz Modula-3 (dysponująca zarówno możliwością obsługi, jak i specyfikacji wyjątków).

Liskov i Snyder w swej publikacji[8] poświęconej temu zagadnieniu zauważają, że podstawową wadą takich języków jak C, zgłaszających informacje o błędach w sposób, który nie wymusza obsługi, jest:

> *„... po każdym wywołaniu należy umieścić instrukcję warunkową sprawdzającą zwrócony wynik. Wymaganie to prowadzi do powstawania programów, których kod jest trudny do analizy i prawdopodobnie także nieefektywny, co zniechęca programistów do zgłaszania i obsługi wyjątków."*

Należy zauważyć, że początkowym powodem wprowadzania obsługi wyjątków było dążenie do wyeliminowania tego wymogu; niemniej jednak właśnie taki kod należy tworzyć w Javie w przypadku obsługi wyjątków sprawdzanych. Autorzy cytowanej wcześniej publikacji kontynuują:

[8] Barbara Liskov i Alan Snyder: *Exception Handling in CLU*, IEEE Transactions on Software Engineering, tom SE-5, nr 6, listopad 1979. Dokument ten nie jest dostępny w internecie, a jedynie w formie drukowanej, a zatem, aby zapoznać się z jego treścią, należy znaleźć jego kopię w bibliotece.

„... kod procedury obsługi musiałby być dołączony do wywołania zgłaszającego wyjątek, co prowadziłoby do powstawania programów, których kod byłby trudny od analizy, a wyrażenia byłyby oddzielone od procedur obsługi wyjątków."

Stroustrup, projektując obsługę wyjątków w języku C++, bazował na rozwiązaniach przyjętych w CLU, przy czym stwierdził, że celem ma być redukcja wielkości kodu koniecznego do naprawienia błędu. Uważam, że zauważył on, że pisząc programy w języku C, programiści zazwyczaj nie tworzyli kodu obsługującego błędy, gdyż zarówno wielkość, jak i umiejscowienie tego kodu było zniechęcające i mogło rozpraszać. Poza tym programiści byli przyzwyczajeni do wykorzystywania rozwiązań charakterystycznych dla pisania programów w języku C — całkowicie ignorowali informacje o błędach i używali programów uruchomieniowych do lokalizowania problemów. Aby programiści C zaczęli stosować wyjątki, należało ich przekonać do pisania „dodatkowego" kodu, który wcześniej nie był stosowany. A zatem, aby przekonać ich do stosowania tego lepszego sposobu obsługi błędów, dodatkowy kod nie mógł być zbyt uciążliwy. Sądzę, że koniecznie należy pamiętać o tym założeniu, analizując kod Javy używany do obsługi wyjątków sprawdzanych.

W C++ została wykorzystana jeszcze jedna idea zapożyczona z języka CLU: specyfikacja wyjątku, stosowana do programowego zaznaczenia w sygnaturze metody, jaki wyjątek może zostać zwrócony. Zastosowanie specyfikacji wyjątku służy w rzeczywistości dwóm celom. Może ona informować: „Ja zgłaszam ten wyjątek w swoim kodzie, a Ty go obsługujesz". Lecz jednocześnie może także mieć drugie znaczenie: „Ignoruję ten wyjątek wykonywania mojego kodu, który został zgłoszony, a Ty powinieneś go obsłużyć". Omawiając mechanizmy oraz składnię obsługi wyjątków, koncentrowaliśmy się na części „Ty masz ją obsłużyć". Jednak w tym przypadku szczególnie interesuje mnie to, że często ignorujemy wyjątki, a ich specyfikowanie może to wykryć.

W języku C++ specyfikacja wyjątków nie należy do informacji o typie funkcji. Jedynym testem wykonywanym podczas kompilacji jest sprawdzenie i zagwarantowanie spójności specyfikacji wyjątków. Na przykład, jeśli funkcja lub metoda zgłaszają wyjątki, to jej przeciążone lub przesłonięte wersje także muszą zgłaszać te wyjątki. Jednak, w odróżnieniu od Javy, kompilator nie sprawdza, czy metoda bądź funkcja faktycznie zgłosi wyjątek, albo czy specyfikacja wyjątków jest kompletna (czyli czy zostały w niej uwzględnione wszystkie wyjątki, które metoda może zgłosić). Sprawdzenie to następuje natomiast podczas działania programu. Jeśli zostanie zgłoszony wyjątek niezgodny ze specyfikacją, to program napisany w C++ wywoła standardową funkcję biblioteczną o nazwie `unexpected()`.

Ciekawe, że ze względu na wykorzystanie wzorców specyfikacje wyjątków w ogóle nie są stosowane w standardowej bibliotece języka C++. W Javie mamy zaś do czynienia z ograniczeniami co do sposobu używania typów ogólnych ze specyfikacjami wyjątków.

Perspektywy

W pierwszej kolejności warto zauważyć, że w Javie w efektywny sposób wymyślono wyjątki sprawdzane (nie ma wątpliwości, że zostały one zainspirowane specyfikacjami wyjątków stosowanymi w C++ oraz faktem, że programiści używający tego języka zazwyczaj całkowicie je ignorowali). Stanowią one eksperyment, którego jak dotąd nie zdecydowali się powtórzyć twórcy żadnego innego języka.

Po drugie, podstawowe przykłady i niewielkie programy pokazują, że sprawdzane wyjątki są bezdyskusyjnie bardzo dobrym rozwiązaniem. Pojawiały się sugestie, że w przypadku pisania dużych programów sprawdzane wyjątki przysparzają nieznacznych problemów. Oczywiście program nie staje się duży w jednej chwili — proces rozrastania się kodu trwa długo i zaczyna się niezauważenie. Języki nieprzystosowane do tworzenia dużych projektów mogą być stosowane do pisania małych projektów, które się rozrastają, a ich twórcy w pewnym momencie zdają sobie sprawę z tego, że kod, który wcześniej był łatwy do zarządzania, teraz stał się trudny. Uważam, że właśnie to może się zdarzyć w przypadku zbyt rozbudowanego sprawdzania typów, a w szczególności w przypadku korzystania z wyjątków sprawdzanych.

Także wielkość programu wydaje się mieć duże znaczenie. Jest to problem, gdyż większość prezentowanych zagadnień jest omawiana na przykładzie niewielkich programów. Jeden z projektantów języka C# zauważył:

> *„Analiza niewielkich programów prowadzi do wniosków, że wymóg podawania specyfikacji wyjątków może zarówno poprawić efektywność programistów, jak i jakość kodu; jednak doświadczenia z dużymi projektami programistycznymi sugerują coś zupełnie przeciwnego — spadek efektywności i niewielką poprawę jakości lub jej całkowity brak"*[9].

Oto wypowiedź twórców CLU dotycząca nieprzechwyconych wyjątków:

> *„Uważaliśmy, że zmuszanie programistów do tworzenia procedur obsługi wyjątków w sytuacjach, w których nie można wykonać żadnej sensownej czynności jest nierealistyczne."*

Wyjaśniając, dlaczego deklaracja funkcji, której nie towarzyszy żadna specyfikacja, oznacza, że funkcja ta może zgłaszać *dowolny* wyjątek, a nie że nie może zgłaszać *żadnego*, Stroustrup stwierdził:

> *„Jednak to wymagałoby podawania specyfikacji wyjątków w niemal wszystkich funkcjach, stanowiłoby poważny powód rekompilacji i uniemożliwiło współdziałanie z oprogramowaniem napisanym w innych językach. To z kolei zachęciłoby programistów do odrzucenia mechanizmów obsługi wyjątków i pisania kodu pomijającego wyjątki. W ten sposób programiści, którzy nie zauważyliby wyjątku, zyskiwaliby fałszywe poczucie bezpieczeństwa."*[10]

Dokładnie to samo zachowanie — pomijanie obsługi wyjątków — można zauważyć, obserwując postępowanie programistów z wyjątkami sprawdzanymi w Javie.

Martin Fowler (autor książki *UML Distilled*, *Refactoring* oraz *Analysis Patterns*) napisał w liście do mnie:

> *„... ogólnie uważam, że wyjątki są dobre, jednak sprawdzane wyjątki w Javie przysparzają więcej problemów niż korzyści."*

---

[9] *http://discuss.develop.com/archives/wa.exe?A2=ind0011A&L=DOTNET&P=R32820*

[10] Bjarne Stroustrup, *The C++ Programming Language, 3rd edition*, Addison-Wesley 1997, strona 376.

Obecnie uważam, że znaczącym postępem dokonanym w Javie było ujednolicenie modelu raportowania błędów, polegającego na tym, że informacje o wszystkich błędach są *zgłaszane* przy wykorzystaniu wyjątków. Ze względu na zgodność z językiem C, w którym można było ignorować wszystkie błędy, rozwiązania tego nie można było zastosować w C++. Niemniej jednak, dysponując spójnym systemem informowania o błędach przy użyciu wyjątków, można tych wyjątków używać wtedy, gdy jest to pożądane, a w pozostałych przypadkach będą one propagować aż do najwyższego poziomu (konsoli lub programu zewnętrznego). Gdy model obsługi wyjątków języka C++ został zmieniony w Javie, tak że wszystkie błędy są zgłaszane za pomocą wyjątków, dodatkowe wykorzystanie sprawdzanych wyjątków nie jest już tak niezbędne.

Dawniej głęboko wierzyłem, że zarówno sprawdzane wyjątki, jak i statyczna kontrola typów są istotnymi czynnikami mającymi wpływ na tworzenie solidnych programów. Jednak zarówno zasłyszane informacje, jak i bezpośrednie doświadczenia[11] z językami, które są bardziej dynamiczne niż statyczne, sprawiły, że aktualnie uważam, że największe korzyści dają:

1. Ujednolicony model zgłaszania informacji o błędach bazujący na wykorzystaniu wyjątków, niezależnie od tego, czy programista jest zmuszony przez kompilator do ich obsługi czy też nie.
2. Sprawdzanie typów, niezależnie od tego, *kiedy* są one sprawdzane. Oznacza to, że o ile tylko zostanie wymuszone użycie odpowiedniego typu, to fakt, czy nastąpi ono podczas kompilacji czy podczas wykonywania programu, nie ma większego znaczenia.

Co więcej, ograniczenie wymagań sprawdzanych podczas kompilacji kodu w znaczący sposób poprawia efektywność programistów. I faktycznie, *mechanizm refleksji* (a w końcu także *typy ogólne*) są wymagane, by zrekompensować nadmiernie ograniczającą statyczną kontrolę typów; przekonasz się o tym w następnych rozdziałach oraz w wielu przykładach przedstawionych w niniejszej książce.

Niektórzy zarzucali mi już, że to bluźnierstwo, że głosząc takie opinie, zniszczę swoją reputację, cywilizacja legnie w gruzach i więcej projektów programistycznych zakończy się niepowodzeniem. Wiara, że kompilator zgłaszający błędy podczas kompilacji programów może doprowadzić do pomyślnego zakończenia projektu, jest ważna, niemniej jednak jeszcze ważniejsze jest uświadomienie sobie, że możliwości kompilatora są ograniczone. W suplemencie publikowanym pod adresem *http://MindView.net/Books/BetterJava* zwracam uwagę na znaczenie zautomatyzowanego procesu konstruowania programów i testowania jednostkowego; mechanizmy te są znacznie bardziej przydatne niż próba zamienienia wszelkich problemów na błędy syntaktyczne. Warto pamiętać, że:

> *dobry język programowania to taki język, który pomaga programistom tworzyć dobre programy. Żaden język nie uniemożliwi swym użytkownikom pisania złych programów*[12].

[11] Pośrednie z językiem Smalltalk uzyskane dzięki korespondencji z wieloma doświadczonymi programistami używającymi tego języka, bezpośrednie z językiem Python (*www.python.org*).

[12] Kees Koster, projektant języka CDL zacytowany przez Bertranda Meyera projektanta języka Eiffel.

W każdym razie prawdopodobieństwo, że wyjątki sprawdzane zostaną usunięte z Javy, jest bardzo małe. Byłaby to zbyt drastyczna zmiana języka, choć jej zwolennicy w firmie Sun wydają się być całkiem liczni. Firma Sun prowadziła w przeszłości i wciąż prowadzi politykę całkowitej wstecznej zgodności — abyś zrozumiał, o co chodzi, wystarczy powiedzieć, że niemal całe oprogramowanie stworzone w tej firmie działa na dowolnym wyprodukowanym przez nią sprzęcie komputerowym, niezależnie od tego, kiedy został on stworzony. Niemniej jednak, jeśli zauważysz, że jakieś sprawdzane wyjątki zaczynają Ci przeszkadzać, a w szczególności jeśli zauważysz, że jesteś zmuszony do przechwytywania wyjątków, choć nie wiesz, co z nimi zrobić, wiedz, że istnieją alternatywne rozwiązania.

Przekazywanie wyjątków na konsolę

W prostych programach, takich jak wiele spośród przykładów przedstawianych w niniejszej książce, najprostszym sposobem zachowania wyjątków bez konieczności pisania rozbudowanego kodu jest przekazywanie ich poza metodę `main()` na konsolę. Na przykład, aby otworzyć plik do odczytu (szczegółowe informacje na temat tej operacji poznasz w rozdziale „Wejście-wyjście"), należy otworzyć i zamknąć strumień `FileInputStream`, który zgłasza wyjątki. W prostym programie operację tę można wykonać w sposób przedstawiony niżej (jak się przekonasz, będzie on wykorzystywany w wielu przykładach przedstawionych w tej książce):

```
//: exceptions/MainException.java
import java.io.*;

public class MainException {
  // Przekazywanie wszystkich wyjątków na konsolę:
  public static void main(String[] args) throws Exception {
    // Otwarcie pliku:
    FileInputStream file =
      new FileInputStream("MainException.java");
    // Użycie pliku...
    // Zamknięcie pliku:
    file.close();
  }
} ///:~
```

Należy zauważyć, że specyfikacja wyjątku może się także pojawić w metodzie `main()`, a w takim przypadku metoda ta może zgłosić wyjątek `Exception`, czyli wyjątek klasy, po której dziedziczą wszystkie wyjątki sprawdzane. Przekazując wyjątki z metody `main()` na konsolę, jesteśmy zwolnieni z obowiązku umieszczania w jej kodzie bloków `try` oraz `catch`. (Niestety, operacje wejścia-wyjścia są znacznie bardziej złożone, niż sugerowałby to powyższy przykład; dlatego też nie ekscytuj się zbytnio, zanim nie przeczytasz rozdziału „Wejście-wyjście")

Ćwiczenie 26. W programie *MainException.java* zmień łańcuch znaków określający nazwę pliku, tak by wskazywał on na nieistniejący plik. Uruchom program i sprawdź jego wyniki (1).

Zamiana wyjątków sprawdzanych na niesprawdzane

Zgłaszanie wyjątków w metodzie `main()` jest wygodne podczas tworzenia prostych programów dla własnych potrzeb, ale ogólnie rzecz biorąc, nie jest to rozwiązanie przydatne. Prawdziwe problemy pojawiają się podczas pisania kodu zwyczajnych metod, gdy zdamy sobie sprawę z tego, że: „W tym miejscu nie mamy żadnego pomysłu, co zrobić z danym wyjątkiem, a nie chcemy ani go „połykać", ani wyświetlać jakiegoś banalnego komunikatu". Można po prostu umieścić wyjątek sprawdzany „wewnątrz" wyjątku klasy `RuntimeException`, przez przekazanie wyjątku do konstruktora `RuntimeException`, jak tu:

```
try {
  // ... robimy coś przydatnego
} catch (NieWiemCoZrobićZTymSprawdzanymWyjątkiem e) {
  throw new RuntimeException(e);
}
```

Wydaje się, że jest to doskonałe rozwiązanie w sytuacjach, gdy nie chcemy „wyłączać" wyjątku sprawdzanego — zgłoszony wyjątek nie jest „połykany", nie trzeba go umieszczać w specyfikacji wyjątków danej metody, a jednocześnie, dzięki wykorzystaniu sekwencji wyjątków, nie gubimy żadnych informacji na jego temat.

Technika ta stwarza możliwość ignorowania wyjątków i przekazywania ich w górę stosu wywołań bez konieczności tworzenia bloków `try` i `catch` bądź specyfikacji wyjątków. Wciąż jednak, dzięki wykorzystaniu metody `getCause()`, istnieje możliwość przechwycenia i obsługi konkretnego wyjątku. Demonstruje to kolejny przykład:

```
//: exceptions/TurnOffChecking.java
// "Wyłączanie" wyjątków sprawdzanych.
import java.io.*;
import static net.mindview.util.Print.*;

class WrapCheckedException {
  void throwRuntimeException(int type) {
    try {
      switch(type) {
        case 0: throw new FileNotFoundException();
        case 1: throw new IOException();
        case 2: throw new RuntimeException("Gdzie jestem?");
        default: return;
      }
    } catch(Exception e) { // Adaptacja do wyjątku niesprawdzanego:
      throw new RuntimeException(e);
    }
  }
}

class SomeOtherException extends Exception {}

public class TurnOffChecking {
  public static void main(String[] args) {
    WrapCheckedException wce = new WrapCheckedException();
    // Metodę throwRuntimeException() można wywołać bez bloku try
    // i pozwolić na opuszczenie metody przez wyjątek RuntimeException:
    wce.throwRuntimeException(3);
    // Albo przechwycić wyjątki:
```

```
    for(int i = 0; i < 4; i++)
      try {
        if(i < 3)
          wce.throwRuntimeException(i);
        else
          throw new SomeOtherException();
      } catch(SomeOtherException e) {
          print("SomeOtherException: " + e);
      } catch(RuntimeException re) {
        try {
          throw re.getCause();
        } catch(FileNotFoundException e) {
          print("FileNotFoundException: " + e);
        } catch(IOException e) {
          print("IOException: " + e);
        } catch(Throwable e) {
          print("Throwable: " + e);
        }
      }
  }
} /* Output:
FileNotFoundException: java.io.FileNotFoundException
IOException: java.io.IOException
Throwable: java.lang.RuntimeException: Gdzie jestem?
SomeOtherException: SomeOtherException
*///:~
```

Metoda `WrapCheckedException.throwRuntimeException()` zawiera kod generujący różne typy wyjątków. Są one następnie przechwytywane i „umieszczane wewnątrz" obiektów `RuntimeException`, przez co stają się ich „przyczyną".

W powyższym przykładzie widać, że metodę `throwRuntimeException()` można wywołać bez konieczności stosowania bloku `try`, gdyż nie zgłasza ona żadnych wyjątków sprawdzanych. Jednak w momencie gdy będziemy gotowi do przechwycenia wyjątków, wciąż możemy przechwycić dowolny z nich, umieszczając tworzony kod wewnątrz bloku `try`. W pierwszej kolejności należy przechwycić wyjątki, które może zgłosić kod umieszczony w bloku `try`; w powyższym przykładzie takim wyjątkiem jest `SomeOtherException`. Natomiast w ostatniej kolejności należy przechwycić wyjątek `RuntimeException` i zgłosić wyjątek zwrócony przez wywołanie metody `getCause()` (czyli wyjątek „umieszczony wewnątrz" obiektu `RuntimeException`). W ten sposób pobierane są oryginalne wyjątki, które następnie można obsłużyć w odrębnych klauzulach `catch`.

Opisana powyżej technika umieszczania sprawdzanych wyjątków w wyjątkach `RuntimeException` będzie stosowana w niektórych przykładach przedstawionych w dalszej części książki, oczywiście jeśli rozwiązanie to będzie wskazane. Alternatywą byłoby wyprowadzenie własnej podklasy `RuntimeException`. Można by wtedy zaniechać przechwytywania, ale pozostawić tę możliwość innym chętnym.

Ćwiczenie 27. Zmień ćwiczenie 3. tak, aby skonwertować wyjątek na `RuntimeException` (1).

Ćwiczenie 28. Zmodyfikuj ćwiczenie 4. tak, aby własna klasa wyjątku dziedziczyła po `RuntimeException`, i pokaż, że kompilator pozwoli Ci opuścić blok `try` (1).

Ćwiczenie 29. Zmień typy wszystkich wyjątków z programu *StormyInning.java* tak, aby dziedziczyły po `RuntimeException`, i pokaż, że niepotrzebne są wtedy specyfikacje wyjątków i bloki `try`. Usuń z kodu komentarze „//!" i pokaż, że oznaczone nimi metody da się skompilować mimo braku specyfikacji wyjątków (1).

Ćwiczenie 30. Zmodyfikuj program *Human.java* tak, aby wyjątki dziedziczyły po `RuntimeException`. Zmień metodę `main()` tak, aby do obsługi różnych typów wyjątków wykorzystać w niej technikę z programu *TurnOffChecking.java* (2).

Wskazówki

Wyjątków należy używać do:

1. Naprawiania problemów na odpowiednim poziomie (należy unikać przechwytywania wyjątków, jeśli nie wiadomo co z nimi zrobić).
2. Naprawienia problemu i ponownego wywołania metody, która spowodowała wyjątek.
3. Wyjścia z sytuacji, która spowodowała wyjątek, i kontynuowania bez ponownego wywołania szwankującej metody.
4. Wygenerowania alternatywnego rozwiązania zamiast tego, które miała wyprodukować metoda.
5. Zrobienia, co tylko się da w aktualnym kontekście, i zgłoszenia ponownie *tego samego* wyjątku do kontekstu nadrzędnego.
6. Zrobienia, co tylko się da w aktualnym kontekście, i zgłoszenia *innego* wyjątku do kontekstu nadrzędnego.
7. Zakończenia programu.
8. Upraszczania (jeśli schemat obsługi wyjątków sprawia, że wszystko jest jeszcze bardziej skomplikowane, to jest on żmudny i irytujący w użyciu).
9. Sprawiania, aby biblioteka i program były bezpieczniejsze (jest to inwestycja krótkoterminowa przy poprawianiu błędów oraz długofalowa dla poprawienia niezawodności aplikacji).

Podsumowanie

Wyjątki są integralnym elementem programowania w języku Java; brak umiejętności ich stosowania znacząco ogranicza możliwości programisty. Z tego powodu wyjątki pojawiły się właśnie w tym miejscu książki, to znaczy jeszcze przed licznymi bibliotekami, które intensywnie korzystają z wyjątków (jak zapowiadana już biblioteka wejścia-wyjścia).

Jedną z zalet mechanizmu wyjątków jest możliwość skupienia rozwiązywanego problemu w jednym miejscu kodu, a obsługi ewentualnych błędów w innym. I choć wyjątki opisuje się zasadniczo jako narzędzia do zgłaszania błędów i przywracania stanu programu operujące w czasie wykonania, zastanawiam się, jak często faktycznie dochodzi do implementacji owego przywracania stanu. Moim zdaniem to mniej niż 10 procent przypadków, a nawet w tych nielicznych sytuacjach wznowienie polega zazwyczaj na odwinięciu stosu do ostatniego znanego stanu stabilnego bez podejmowania faktycznego wysiłku wznawiania działalności programu możliwie blisko wykrycia błędu. Osobiście uważam, że prawdziwa przydatność wyjątków tkwi jednak w informowaniu o błędach. Fakt, że Java nalega, aby wszelkie błędy były sygnalizowane w postaci wyjątków, to wielka przewaga wobec takich języków jak C++, gdzie błędy zwykło się sygnalizować na wiele różnych sposobów, a niekiedy wcale. Spójny system raportowania o błędach oznacza brak konieczności ciągłego zadawania pytania o błędy przeciskające się przez szczeliny programu przy każdym pisanym kawałku kodu (o ile się pamięta, aby nie „połykać" wyjątków!).

W następnych rozdziałach przekonasz się, że odłożenie tego pytania na bok — nawet jeśli odbywa się to przy użyciu `RuntimeException` — daje komfort skupiania się na ciekawszych, najbardziej istotnych aspektach danego projektu i jego implementacji.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 13.

Ciągi znaków

Manipulowanie ciągami znaków to bez wątpienia jedna z najbardziej typowych czynności w programowaniu.

Dotyczy to zwłaszcza systemów WWW, które z kolei są domeną języka Java. Dlatego w tym rozdziale zajmiemy się klasą, która z pewnością jest najpowszechniej używaną klasą języka — klasą `String`, oraz klasami względem niej pomocniczymi i narzędziowymi.

Niezmienność ciągów znakowych

Obiekty klasy `String` są niezmienne (ang. *immutable*). Jeśli przejrzysz dokumentację JDK dla tej klasy, przekonasz się, że każda z jej metod, która miałaby modyfikować ciąg znaków, w istocie zwraca całkiem nowy obiekt `String` z ciągiem wynikowym. Pierwotny egzemplarz `String` pozostaje nietknięty.

Spójrzmy na poniższy kod:

```
//: strings/Immutable.java
import static net.mindview.util.Print.*;

public class Immutable {
  public static String upcase(String s) {
    return s.toUpperCase();
  }
  public static void main(String[] args) {
    String q = "Jak leci?";
    print(q); // Jak leci?
    String qq = upcase(q);
    print(qq); // JAK LECI?
    print(q); // Jak leci?
  }
} /* Output:
Jak leci?
JAK LECI?
Jak leci?
*///:~
```

Przy przekazywaniu `q` do metody `upcase()` tworzona jest kopia referencji `q`. Fizyczny obiekt docelowy referencji pozostaje w swoim pierwotnym położeniu. Przekazywanie ciągów polega więc na kopiowaniu referencji.

W definicji metody upcase() widać, że przekazywana do niej referencja posiada tam nazwę s i istnieje jedynie tak długo, jak długo wykonywane jest ciało metody upcase(). Po zakończeniu metody lokalna referencja s znika. Metoda zwraca wynik, którym jest pierwotny ciąg z literami zamienionymi na ich wielkie odpowiedniki. Oczywiście faktycznie dochodzi tu do zwrócenia referencji wyniku, okazuje się jednak, że referencja odnosi się już do zupełnie nowego obiektu, a pierwotny ciąg wskazywany referencją q pozostaje nietknięty.

Zwykle takie zachowanie jest jak najbardziej pożądane. Załóżmy, że napiszesz:

```
String s = "asdf";
String x = Immutable.upcase(s);
```

Czy na pewno wywołanie upcase() ma zmienić argument? Dla czytającego kod argument jest zazwyczaj elementem informacji przekazywanym do metody (elementem sterującym jej wykonaniem), a nie podmiotem modyfikacji. Niezmienność ciągów znaków jest więc o tyle ważna, że ułatwia pisanie, czytanie i ogarnięcie kodu.

StringBuilder kontra przeciążony operator '+'

Skoro obiekty String są niezmienne, można danemu egzemplarzowi nadawać dowolnie wielką liczbę referencji. Ponieważ String jest obiektem tylko do odczytu, nie zachodzi ryzyko, że za pośrednictwem jednej z referencji dojdzie do zmiany, która ujawni się w pozostałych referencjach.

Niezmienność może jednak ujemnie wpływać na efektywność. Tak jest w przypadku operatora '+', który został przeciążony dla obiektów klasy String. Przeciążenie oznacza, że operacja otrzymuje inne od pierwotnego znaczenie, właściwe dla typu operandów (operatory '+' i '+=' dla klasy String to jedyne przykłady przeciążenia operatorów w języku Java — Java nie pozwala programiście na przeciążanie żadnych innych operatorów[1]).

Operator '+' pozwala na scalanie (konkatenację) ciągów reprezentowanych obiektami String:

```
//: strings/Concatenation.java

public class Concatenation {
  public static void main(String[] args) {
    String mango = "mango";
    String s = "abc" + mango + "def" + 47;
    System.out.println(s);
```

[1] W C++ programista może przeciążać operatory do woli. Ponieważ jest to często proces złożony (zobacz rozdział 10. książki *Thinking in C++, 2nd Edition*, Prentice Hall, 2000), projektanci języka Java uznali tę możliwość za niepożądaną i wykluczyli z języka. Szkoda jedynie, że sami nie potrafili się bez przeciążania obejść, a do tego przeciążanie operatorów w Javie byłoby dalece prostsze, niż w C++. Dowód tego można znaleźć w językach Python (*www.Python.org*) i C# — oba posiadają mechanizm odśmiecania pamięci i dają możliwość łatwego przeciążania operatorów.

```
  }
} /* Output:
abcmangodef47
*///:~
```

Łatwo sobie wyobrazić, jak *mogłoby* to działać. Nienazwany egzemplarz `String` z ciągiem „abc" mógłby posiadać metodę `append()`, która tworzyłaby nowy egzemplarz z ciągiem „abc" scalonym z zawartością egzemplarza `mango`. Wynikowy obiekt `String` mógłby potem analogicznie skonstruować egzemplarz z ciągiem uzupełnionym o „def" i tak dalej.

Działałoby to poprawnie, ale wymagałoby tworzenia całego mnóstwa egzemplarzy klasy `String` tylko po to, aby z nich składać coraz większe ciągi — w programie powstałoby więc mnóstwo tymczasowych, roboczych obiektów `String`, które niemal natychmiast po powstaniu oczekiwałyby już tylko na odśmiecenie. Podejrzewam, że początkowo architekci języka próbowali właśnie tego rozwiązania (co należy potraktować jako nauczkę z dziedziny projektowania oprogramowania — o systemie nie wiadomo tak naprawdę prawie nic, dopóki nie wypróbuje się go w boju). Podejrzewam też, że szybko wpadli na trop niedopuszczalnego marnotrawstwa.

Aby sprawdzić, co tak naprawdę dzieje się przy konkatenacji, należałoby zdekompilować powyższy kod przy użyciu narzędzia *javap*, wchodzącego w skład JDK. Oto stosowne polecenie:

```
javap -c Concatenation
```

Opcja `-c` wymusza wygenerowanie kodów bajtowych dla maszyny wirtualnej. Po pozbyciu się rzeczy mało ciekawych i wstępnej obróbce edycyjnej dojdziemy do sedna:

```
public static void main(java.lang.String[]);
  Code:
   Stack=2, Locals=3, Args_size=1
   0:  ldc  #2; //String mango
   2:  astore_1
   3:  new  #3; //class StringBuilder
   6:  dup
   7:  invokespecial  #4; //StringBuilder."<init>":()
   10:  ldc  #5; //String abc
   12:  invokevirtual  #6; //StringBuilder.append:(String)
   15:  aload_1
   16:  invokevirtual  #6; //StringBuilder.append:(String)
   19:  ldc  #7; //String def
   21:  invokevirtual  #6; //StringBuilder.append:(String)
   24:  bipush  47
   26:  invokevirtual  #8; //StringBuilder.append:(I)
   29:  invokevirtual  #9; //StringBuilder.toString:()
   32:  astore_2
   33:  getstatic  #10; //Field System.out.PrintStream;
   36:  aload_2
   37:  invokevirtual  #11; //PrintStream.println:(String;)
   40:  return
```

Kto miał do czynienia z językiem asemblerowym, nie będzie specjalnie zaszokowany — instrukcje w rodzaju `dup` czy `invokevirtual` to odpowiedniki instrukcji języka asemblerowego w obrębie wirtualnej maszyny Java. Ci zaś, którzy nie wiedzą, co to język

asemblerowy, nie muszą się w ogóle tym przejmować — ważne, żeby dostrzegli zaangażowanie przez kompilator klasy `java.lang.StringBuilder`. W kodzie źródłowym nie było o niej żadnej wzmianki, ale kompilator i tak zdecydował o jej użyciu, bo efektywnie realizuje konkatenację ciągów znaków.

W tym przypadku kompilator utworzył obiekt `StringBuilder` w celu skonstruowania ciągu `s`, po czym czterokrotnie wywołał na rzecz `s` metodę `append()`, po razie dla każdego kolejnego kawałka scalanego ciągu. Na końcu wywołaniem `toString()` wygenerował wynikowy ciąg, zachowany (instrukcją `astore_2`) w `s`.

Zanim przyjmiesz założenie, że możesz dowolnie stosować obiekty `String`, a kompilator tam, gdzie jest to potrzebne, zastosuje powyższą optymalizację, powinieneś przyjrzeć się nieco uważniej poczynaniom kompilatora. Oto przykład wygenerowania obiektu `String` na dwa sposoby: za pomocą egzemplarzy klasy `String` i przy jawnym, ręcznym wykorzystaniu klasy `StringBuilder`:

```
//: strings/WhitherStringBuilder.java

public class WhitherStringBuilder {
  public String implicit(String[] fields) {
    String result = "";
    for(int i = 0; i < fields.length; i++)
      result += fields[i];
    return result;
  }
  public String explicit(String[] fields) {
    StringBuilder result = new StringBuilder();
    for(int i = 0; i < fields.length; i++)
      result.append(fields[i]);
    return result.toString();
  }
} ///:~
```

Spróbuj ponownie użyć programu javap poleceniem `javap -c WhitherStringBuilder`, aby dotrzeć do (uproszczonego) kodu dwóch powyższych metod. Oto pierwsza z nich, `implicit()`:

```
public java.lang.String implicit(java.lang.String[]);
  Code:
   0:  ldc  #2; //String
   2:  astore_2
   3:  iconst_0
   4:  istore_3
   5:  iload_3
   6:  aload_1
   7:  arraylength
   8:  if_icmpge  38
   11:  new  #3; //class StringBuilder
   14:  dup
   15:  invokespecial  #4; //StringBuilder."<init>":()
   18:  aload_2
   19:  invokevirtual  #5; //StringBuilder.append:(String)
   22:  aload_1
   23:  iload_3
   24:  aaload
```

```
25:  invokevirtual  #5: //StringBuilder.append:(String)
28:  invokevirtual  #6: //StringBuilder.toString:()
31:  astore_2
32:  iinc  3, 1
35:  goto  5
38:  aload_2
39:  areturn
```

Zwróć uwagę na wiersze 8. i 35., które tworzą coś w rodzaju pętli. Instrukcja z wiersza 8. to instrukcja skoku do wiersza 38. pod warunkiem zachodzenia relacji „większe lub równe" dla operandów umieszczonych na stosie. Wiersz 35. to powrót do początku pętli w wierszu 5. Okazuje się, że konstrukcja egzemplarza `StringBuilder` odbywa się *wewnątrz* tej pętli, co oznacza, że każda iteracja generuje nowy obiekt `StringBuilder`.

Oto kod bajtowy metody `explicit()`:

```
public java.lang.String explicit(java.lang.String[]):
  Code:
    0:  new  #3: //class StringBuilder
    3:  dup
    4:  invokespecial  #4: //StringBuilder."<init>":()
    7:  astore_2
    8:  iconst_0
    9:  istore_3
    10:  iload_3
    11:  aload_1
    12:  arraylength
    13:  if_icmpge  30
    16:  aload_2
    17:  aload_1
    18:  iload_3
    19:  aaload
    20:  invokevirtual  #5: //StringBuilder.append:(String)
    23:  pop
    24:  iinc  3, 1
    27:  goto  10
    30:  aload_2
    31:  invokevirtual  #6: //StringBuilder.toString:()
    34:  areturn
```

Kod pętli jest tu nie tylko prostszy i krótszy, ale i metoda tworzy tylko jeden obiekt `StringBuilder`. Jawne utworzenie egzemplarza `StringBuilder` pozwala też na wstępny przydział odpowiedniego rozmiaru (o ile posiadamy informację o rozmiarze wynikowego ciągu), tak aby nie trzeba było wciąż powiększać bufora montażowego.

Dlatego przy definiowaniu własnych wersji metody `toString()`, jeśli operacje na ciągach są na tyle proste, że kompilator może je sam skutecznie zoptymalizować, należałoby zdać się właśnie na kompilator. Ale tam, gdzie w grę wchodzi pętla, lepiej jawnie wykorzystać w metodzie `toString()` obiekt `StringBuilder`, jak tu:

```
//: strings/UsingStringBuilder.java
import java.util.*;

public class UsingStringBuilder {
  public static Random rand = new Random(47);
  public String toString() {
```

```
    StringBuilder result = new StringBuilder("[");
    for(int i = 0; i < 25; i++) {
      result.append(rand.nextInt(100));
      result.append(", ");
    }
    result.delete(result.length()-2, result.length());
    result.append("]");
    return result.toString();
  }
  public static void main(String[] args) {
    UsingStringBuilder usb = new UsingStringBuilder();
    System.out.println(usb);
  }
} /* Output:
[58, 55, 93, 61, 61, 29, 68, 0, 22, 7, 88, 28, 51, 89, 9, 78, 98, 61, 20, 58, 16, 40, 11, 22, 4]
*///:~
```

Zwróć uwagę, że każdy kawałek wyniku jest dodawany do ciągu wywołaniem metody `append()`. Gdybyś pokusił się o skrócenie kodu i napisał `append(a + ":" + c)`, do akcji wkroczyłby kompilator, który w miejsce konkatenacji zastosowałby niejawnie klasę `StringBuilder`.

W przypadku wątpliwości co do wyboru właściwego podejścia zawsze można skorzystać z programu javap i sprawdzić uzyskany efekt.

Choć klasa `StringBuilder` posiada szeroki wachlarz metod, w tym `insert()`, `replace()`, `substring()` i nawet `reverse()`, to najczęściej używa się właśnie `append()` i `toString()`. Zwróć też uwagę na użycie w powyższym przykładzie metody `delete()` w celu usunięcia ostatniego przecinka i spacji przed dodaniem prostokątnego nawiasu zamykającego.

`StringBuilder` to nowość w SE5. Wcześniej w Javie wykorzystywało się klasę `StringBuffer`, która zapewniała bezpieczeństwo w kontekście wielowątkowości (zobacz rozdział „Współbieżność"), co jeszcze bardziej degradowało wydajność. Spodziewam się więc, że operacje na ciągach w Javie SE5 i SE6 będą wyraźnie szybsze.

Ćwiczenie 1. Przeanalizuj metodę `SprinklerSystem.toString()` w pliku *reusing/SprinklerSystem.java* w celu stwierdzenia, czy ewentualna implementacja `toString()` z jawnym obiektem klasy `StringBuilder` pozwoli zmniejszyć liczbę tworzonych egzemplarzy `StringBuilder` (2).

Niezamierzona rekursja

Ponieważ standardowe kontenery Javy (jak każda inna klasa) dziedziczą po klasie `Object`, posiadają metodę `toString()`. Została ona przesłonięta tak, by zwracać ciąg znaków reprezentujący kontener, włączając w to obiekty, które przechowuje. Przykładowo wewnątrz klasy `ArrayList` metoda `toString()` przechodzi przez wszystkie elementy listy i dla każdego wywołuje `toString()`:

```
//: strings/ArrayListDisplay.java
import generics.coffee.*;
import java.util.*;
```

```
public class ArrayListDisplay {
  public static void main(String[] args) {
    ArrayList<Coffee> coffees = new ArrayList<Coffee>();
    for(Coffee c : new CoffeeGenerator(10))
      coffees.add(c);
    System.out.println(coffees);
  }
} /* Output:
[Americano 0, Latte 1, Americano 2, Mocha 3, Mocha 4, Breve 5, Americano 6, Latte 7, Cappuccino 8,
Cappuccino 9]
*///:~
```

Przypuśćmy, że chcielibyśmy w ramach metody `toString()` wypisać adres klasy w pamięci. Wydaje się sensowne po prostu odwołanie się do `this`:

```
//: strings/InfiniteRecursion.java
// Niezamierzona rekurencja.
// {RunByHand}
import java.util.*;

public class InfiniteRecursion {
  public String toString() {
    return "Adres InfiniteRecursion: " + this + "\n";
  }
  public static void main(String[] args) {
    List<InfiniteRecursion> v =
      new ArrayList<InfiniteRecursion>();
    for(int i = 0; i < 10; i++)
      v.add(new InfiniteRecursion());
    System.out.println(v);
  }
} ///:~
```

Jeśli stworzymy obiekt klasy `InfiniteRecursion`, a następnie go wypiszemy, to uzyskamy bardzo długą listę wyjątków. To samo stałoby się, jeśli umieścilibyśmy obiekty `InfiniteRecursion` w kontenerze `ArrayList` i próbowalibyśmy wypisać jego zawartość, jak to widać w przykładzie. To, co się dzieje, to automatyczna konwersja typu na `String`. Kiedy wywołamy:

```
"Adres obiektu typu InfiniteRecursion: " + this
```

kompilator spostrzeże ciąg tekstowy, a za nim + oraz coś, co nie jest typu `String`, toteż usiłuje skonwertować `this` na `String`. Odbywa się to poprzez wywołanie metody `toString()`, co powoduje powstanie rekurencji.

Jeśli rzeczywiście chcielibyśmy wypisać adres obiektu, rozwiązaniem jest wywołanie metody `toString()` z `Object`, która właśnie to robi. Zatem zamiast pisać `this`, powinniśmy użyć wywołania `super.toString()`.

Ćwiczenie 2. Popraw plik *InfiniteRecursion.java* (1).

Operacje na egzemplarzach klasy String

Poniższa tabela wymienia wybrane metody z podstawowego zestawu metod klasy `String`. Metody przeciążone są grupowane we wspólnych wierszach tabeli.

Metoda	Argumenty, przeciążenia	Działanie
Konstruktor	Przeciążenia dla: braku argumentów, klas `String`, `StringBuilder`, `StringBuffer`, tablic `char` i tablic `byte`.	Tworzenie obiektów klasy `String`.
`length()`		Obliczanie liczby znaków w ciągu `String`.
`charAt()`	`int` indeks	Wybieranie znaku (`char`) ze wskazanej pozycji w ciągu `String`.
`getChars()` `getBytes()`	Początek i koniec sekwencji do skopiowania, tablica docelowa, indeks w tablicy docelowej.	Kopiowanie znaków albo bajtów (`byte`) ciągu `String` do tablicy zewnętrznej.
`toCharArray()`		Utworzenie tablicy `char[]` zawierającej komplet znaków z ciągu `String`.
`equals()` `equals-IgnoreCase()`	Ciąg `String` do porównania.	Sprawdzenie równości zawartości dwóch egzemplarzy `String`.
`compareTo()`	Ciąg `String` do porównania.	Zwrócenie wartości ujemnej, zerowej bądź dodatniej w zależności od leksykograficznego uporządkowania ciągu `String` względem argumentu (z rozróżnianiem wielkości liter!).
`contains()`	Sekwencja `CharSequence` do przeszukania.	Zwrócenie `true`, jeśli argument znajduje się w danym ciągu `String`.
`contentEquals()`	Sekwencja `CharSequence` (albo `StringBuffer`) do przeszukania.	Zwrócenie `true`, jeśli argument dokładnie pokrywa się z danym ciągiem `String`.
`EqualsIgnore-Case()`	Ciąg `String` do porównania.	Zwrócenie `true`, jeśli argument dokładnie pokrywa się z danym ciągiem `String` (bez różnicowania wielkości liter).
`regionMatches()`	Przesunięcie w danym ciągu `String`, drugi ciąg `String` i przesunięcie oraz rozmiar podciągu do porównania; wersja przeciążona ignoruje wielkość liter przy porównywaniu.	Zwrócenie wartości `boolean` sygnalizującej pokrywanie się podciągów.
`startsWith()`	Ciąg przedrostka, od którego być może zaczyna się dany obiekt `String`; wersja przeciążona z dodatkowym argumentem przesunięcia, od którego zaczyna się poszukiwanie przedrostka.	Zwrócenie wartości `boolean` sygnalizującej znalezienie podciągu pokrywającego się z argumentem i stanowiącego przedrostek danego ciągu `String`.

Metoda	Argumenty, przeciążenia	Działanie
`endsWith()`	Ciąg `String`, który może stanowić przyrostek danego ciągu `String`.	Zwrócenie wartości `boolean` sygnalizującej znalezienie podciągu pokrywającego się z argumentem i stanowiącego przyrostek danego ciągu `String`.
`indexOf()`, `lastIndexOf()`	Wersje przeciążone dla: znaku `char`, znaku `char` i indeksu początkowego, ciągu `String`, ciągu `String` i indeksu początkowego.	Zwrócenie –1, jeśli argument nie stanowi podciągu danego ciągu `String`; w przeciwnym razie zwrócenie indeksu znaku, od którego rozpoczyna się podciąg identyczny z argumentem.
`substring()` (także `subSequence()`)	Wersje przeciążone: z indeksem początkowym, indeksem początkowym i indeksem końcowym.	Zwrócenie nowego obiektu klasy `String` zawierającego zadany podciąg.
`concat()`	Ciąg `String` do konkatenacji.	Zwrócenie nowego obiektu klasy `String` scalającego zawartość danego ciągu `String` z zawartością argumentu.
`replace()`	Znak do wyszukania, znak zastępczy. Także w wersji wyszukującej i zastępującej całe sekwencje znaków `CharSequence`.	Zwrócenie nowego obiektu `String` z podstawieniami. Jeśli do podstawień nie doszło, metoda zwraca oryginalny obiekt.
`toLowerCase()` `toUpperCase()`		Zwrócenie nowego ciągu `String` z literami zamienionymi na małe (`toLower...`) albo wielkie (`toUpper...`). Jeśli konwersja nie wymaga żadnych zmian, metoda zwraca oryginalny obiekt.
`trim()`		Zwrócenie nowego obiektu `String`, pozbawionego znaków odstępu na początku i końcu zawieranego ciągu. Jeśli oryginalny ciąg nie zawiera takich znaków na początku ani na końcu, metoda zwraca oryginał w nienaruszonym stanie.
`valueOf()`	Przeciążone dla: obiektu `Object`, tablicy `char[]`, tablicy `char[]` i przesunięcia oraz licznika znaków, wartości `boolean`, `char`, `int`, `long`, `float` i `double`.	Zwrócenie ciągu `String` zawierającego znakową reprezentację wartości argumentu.
`intern()`		Wygenerowanie jednej i tylko jednej referencji `String` dla każdej unikatowej sekwencji znaków.

Jak widać, każda metoda klasy `String` pieczołowicie konstruuje nowy obiekt `String` do zwrócenia, chyba że realizowana przez nią operacja nie wymagała modyfikacji ciągu — wtedy metoda zwraca referencję oryginalnego obiektu `String`. W ten sposób unika się narzutu pamięciowego i czasowego.

Metody klasy `String` korzystają szeroko z *wyrażeń regularnych* opisanych w jednym z kolejnych podrozdziałów.

Formatowanie wyjścia

Jedną z długo wyczekiwanych możliwości, które w końcu pojawiły się w wydaniu Java SE5, było formatowanie wyjścia wzorowane na funkcji `printf()` znanej z języka C. Możliwość ta pozwala nie tylko uprościć wypisywanie komunikatów, ale i daje programistom Javy pełnię kontroli w zakresie formatowania i wyrównywania wypisywanych wartości[2].

Funkcja printf()

Funkcja `printf()` z języka C nie składała ciągów, tak jak to się odbywa w języku Java, ale przyjmowała za pośrednictwem argumentu pojedynczy *ciąg formatujący* i wstawiała do niego tekstowe reprezentacje przekazanych wartości, rozmieszczając je zgodnie z żądanym formatem. Zamiast przeciążonego dla ciągów operatora '+' (którego w C nie było) funkcja `printf()` wykorzystuje specjalne symbole zastępcze, kodujące miejsce osadzania wartości w ciągu. Wartości przeznaczone do wstawienia w miejsce tych symboli są przekazywane w postaci listy wartości oddzielanych przecinkami.

Oto przykład:

```
printf("Wiersz 1: [%d %f]\n", x, y);
```

W czasie wykonania w miejsce `%d` ma zostać wstawiona tekstowa reprezentacja wartości `x`, a w miejsce `%f` — tekstowa reprezentacja wartości `y`. Symbole zastępcze noszą miano *specyfikatorów formatu*; nie tylko oznaczają one miejsce wstawienia wartości, ale i sugerują rodzaj tej wartości i tym samym sposób jej formatowania. Na przykład `%d` sugeruje, że `x` jest liczbą całkowitą, a `%f` oznacza, że `y` to wartość zmiennoprzecinkowa (wartość typu `double` lub `float`).

System.out.format()

Java SE5 wprowadziła do użycia metodę `format()`, dostępną dla obiektów klas `PrintStream` i `PrintWriter` (o których dowiesz się więcej w rozdziale „Wejście-wyjście"), do których zalicza się również obiekt `System.out`. Metoda `format()` jest wzorowana na funkcji `printf()` z języka C; ba, co bardziej nostalgiczni programiści mogą wręcz korzystać z metody `printf()`, która deleguje wywołanie właśnie do metody `format()`. Oto prosty przykład jej użycia:

```
//: strings/SimpleFormat.java

public class SimpleFormat {
  public static void main(String[] args) {
    int x = 5;
```

[2] Przy tworzeniu tego podrozdziału oraz podrozdziału „Skanowanie wejścia" współpracował ze mną Mark Welsh.

```
    double y = 5.332542;
    // Klasycznie:
    System.out.println("Wiersz 1: [" + x + " " + y + "]");
    // Współcześnie:
    System.out.format("Wiersz 1: [%d %f]\n", x, y);
    // albo
    System.out.printf("Wiersz 1: [%d %f]\n", x, y);
  }
} /* Output:
Wiersz 1: [5 5,332542]
Wiersz 1: [5 5,332542]
Wiersz 1: [5 5,332542]
*///:~
```

Widać, że metody `printf()` i `format()` są sobie równoważne. W obu przypadkach komunikat wyjściowy składany jest z pojedynczego ciągu formatującego i listy wartości — po jednej dla każdego specyfikatora formatu.

Klasa Formatter

Całość nowych funkcji formatujących jest obsługiwana przez klasę `Formatter` z pakietu `java.util`. Klasę tę możemy traktować jako tłumacza, który zamienia ciąg formatujący i zestaw wartości na ciąg wynikowy. Przy tworzeniu obiektu klasy `Formatter` można określić gdzie ma być zwracany rezultat, przekazując odpowiednią informację do konstruktora:

```
//: strings/Turtle.java
import java.io.*;
import java.util.*;

public class Turtle {
  private String name;
  private Formatter f;
  public Turtle(String name, Formatter f) {
    this.name = name;
    this.f = f;
  }
  public void move(int x, int y) {
    f.format("Żółw %s na pozycji (%d,%d)\n", name, x, y);
  }
  public static void main(String[] args) {
    PrintStream outAlias = System.out;
    Turtle tommy = new Turtle("Tommy",
      new Formatter(System.out));
    Turtle terry = new Turtle("Terry",
      new Formatter(outAlias));
    tommy.move(0,0);
    terry.move(4,8);
    tommy.move(3,4);
    terry.move(2,5);
    tommy.move(3,3);
    terry.move(3,3);
  }
} /* Output:
Żółw Tommy na pozycji (0,0)
Żółw Terry na pozycji (4,8)
```

```
Żółw Tommy na pozycji (3,4)
Żółw Terry na pozycji (2,5)
Żółw Tommy na pozycji (3,3)
Żółw Terry na pozycji (3,3)
*///:~
```

Komunikaty o żółwiu `Tommy` są przekazywane do obiektu `System.out`, a wieści o jego koledze — do aliasu tego obiektu. Konstruktor klasy `Formatter` został przeciążony wersją przyjmującą najróżniejsze lokacje wyjściowe, ale najprzydatniejsze z nich to obiekty klas `PrintStream` (powyżej) i `OutputStream` i `File`. Wrócimy do tego w rozdziale „Wejście-wyjście".

Ćwiczenie 3. Zmodyfikuj program *Turtle.java* tak, aby wysyłał wyjście do `System.err` (1).

W ostatnim przykładzie pojawił się nowy specyfikator formatu w postaci '`%s`'. To symbol zastępczy dla ciągu znaków `String` i jest bodaj najprostszym specyfikatorem obejmującym jedynie typ konwersji.

Specyfikatory formatu

Aby mieć wpływ na wyrównanie wartości i dopełnianie pól przy wyprowadzaniu danych, trzeba uzupełnić specyfikatory formatu dodatkowymi elementami. Oto ogólna składnia specyfikatora:

```
%[indeks_argumentu$][znaczniki][szerokość][.precyzja]konwersja
```

Często zachodzi potrzeba określania minimalnego rozmiaru pola wartości. Służy do tego element *szerokość*. Klasa `Formatter` gwarantuje, że pole wyjściowe będzie miało szerokość równą zadanej liczbie znaków, a wartości niewypełniające całkowicie pola będą uzupełniane spacjami. Domyślnie wartości są wyrównywane do prawej krawędzi pola, ale wyrównanie można zmienić na przeciwne znakiem '`-`' w elemencie znaczników.

Niejako odwrotnością szerokości pola jest *precyzja* określająca maksymalną liczbę znaków wypisywanych wartości. Podczas gdy szerokość pola ma dla wszystkich typów konwersji identyczne znacznie, pojęcie precyzji zależy od typu formatowanej wartości. W przypadku obiektów klasy `String` precyzja to limit liczby znaków z ciągu reprezentowanego obiektem, które zostaną umieszczone w polu wyjściowym. W przypadku wartości zmiennoprzecinkowych precyzja określa liczbę wypisywanych cyfr dziesiętnych (domyślnie 6) z zaokrąglaniem wartości, których nie da się dokładnie wyrazić przy danej precyzji albo z uzupełnianiem wartości zerami z przodu. Ponieważ wartości całkowite nie posiadają części ułamkowej, w ich przypadku precyzja nie ma zastosowania — określenie precyzji dla konwersji typu całkowitego spowoduje zgłoszenie wyjątku.

Poniższy przykład pokazuje użycie specyfikatorów formatu w wypisywaniu paragonu sklepowego:

```
//: strings/Receipt.java
import java.util.*;

public class Receipt {
  private double total = 0;
  private Formatter f = new Formatter(System.out);
  public void printTitle() {
```

```
    f.format("%-15s %5s %10s\n", "Towar", "Ilość", "Cena");
    f.format("%-15s %5s %10s\n", "-----", "-----", "----");
  }
  public void print(String name, int qty, double price) {
    f.format("%-15.15s %5d %10.2f\n", name, qty, price);
    total += price;
  }
  public void printTotal() {
    f.format("%-15s %5s %10.2f\n", "Podatek", "", total*0.22);
    f.format("%-15s %5s %10s\n", "", "", "-----");
    f.format("%-15s %5s %10.2f\n", "Razem", "",
      total * 1.22);
  }
  public static void main(String[] args) {
    Receipt receipt = new Receipt();
    receipt.printTitle();
    receipt.print("Magiczna fasola", 4, 4.25);
    receipt.print("Ziarnko grochu", 3, 5.1);
    receipt.print("Kij Samobij", 1, 14.29);
    receipt.printTotal();
  }
} /* Output:
Towar          Ilość      Cena
-----          -----      ----
Magiczna fasola    4       4,25
Ziarnko grochu     3       5,10
Kij Samobij        1      14,29
Podatek                   5,20
                         -----
Razem                    28,84
*///:~
```

Jak widać, klasa `Formatter` pozwala na szczegółowe sterowanie rozmieszczeniem i wyrównaniem wartości na wyjściu, i to przy zachowaniu zwięzłości specyfikatorów.

Ćwiczenie 4. Zmień plik *Receipt.java* tak, aby szerokość pól była kontrolowana pojedynczym zbiorem stałych; chodzi o to, aby można było łatwo zmieniać szerokość pól przez zmianę wartości przypisanej do stałej (3).

Konwersje

Oto lista najczęściej stosowanych konwersji w specyfikatorach formatu:

Specyfikator	Znaczenie
`d`	Wartość całkowita (w zapisie dziesiętnym).
`c`	Znak Unicode.
`b`	Wartość logiczna (`boolean`).
`s`	Ciąg znaków (`String`).
`f`	Wartość zmiennoprzecinkowa (w zapisie dziesiętnym).
`e`	Wartość zmiennoprzecinkowa (w zapisie naukowym).
`x`	Wartość całkowita (w zapisie szesnastkowym).
`h`	Skrót (ang. *hash code*, w zapisie szesnastkowym).
`%`	Literał „%”.

A to przykład wykorzystania wymienionych konwersji:

```
// strings/Conversion.java
import java.math.*;
import java.util.*;

public class Conversion {
  public static void main(String[] args) {
    Formatter f = new Formatter(System.out);

    char u = 'a';
    System.out.println("u = 'a'");
    f.format("s: %s\n", u);
    // f.format("d: %d\n", u);
    f.format("c: %c\n", u);
    f.format("b: %b\n", u);
    // f.format("f: %f\n", u);
    // f.format("e: %e\n", u);
    // f.format("x: %x\n", u);
    f.format("h: %h\n", u);

    int v = 121;
    System.out.println("v = 121");
    f.format("d: %d\n", v);
    f.format("c: %c\n", v);
    f.format("b: %b\n", v);
    f.format("s: %s\n", v);
    // f.format("f: %f\n", v);
    // f.format("e: %e\n", v);
    f.format("x: %x\n", v);
    f.format("h: %h\n", v);

    BigInteger w = new BigInteger("50000000000000");
    System.out.println(
      "w = new BigInteger(\"50000000000000\")");
    f.format("d: %d\n", w);
    // f.format("c: %c\n", w);
    f.format("b: %b\n", w);
    f.format("s: %s\n", w);
    // f.format("f: %f\n", w);
    // f.format("e: %e\n", w);
    f.format("x: %x\n", w);
    f.format("h: %h\n", w);

    double x = 179.543;
    System.out.println("x = 179.543");
    // f.format("d: %d\n", x);
    // f.format("c: %c\n", x);
    f.format("b: %b\n", x);
    f.format("s: %s\n", x);
    f.format("f: %f\n", x);
    f.format("e: %e\n", x);
    // f.format("x: %x\n", x);
    f.format("h: %h\n", x);

    Conversion y = new Conversion();
    System.out.println("y = new Conversion()");
    // f.format("d: %d\n", y);
    // f.format("c: %c\n", y);
```

```
        f.format("b: %b\n", y);
        f.format("s: %s\n", y);
        // f.format("f: %f\n", y);
        // f.format("e: %e\n", y);
        // f.format("x: %x\n", y);
        f.format("h: %h\n", y);

        boolean z = false;
        System.out.println("z = false");
        // f.format("d: %d\n", z);
        // f.format("c: %c\n", z);
        f.format("b: %b\n", z);
        f.format("s: %s\n", z);
        // f.format("f: %f\n", z);
        // f.format("e: %e\n", z);
        // f.format("x: %x\n", z);
        f.format("h: %h\n", z);
    }
} /* Output: (Sample)
u = 'a'
s: a
c: a
b: true
h: 61
v = 121
d: 121
c: y
b: true
s: 121
x: 79
h: 79
w = new BigInteger("50000000000000")
d: 50000000000000
b: true
s: 50000000000000
x: 2d79883d2000
h: 8842a1a7
x = 179.543
b: true
s: 179.543
f: 179,543000
e: 1.795430e+02
h: 1ef462c
y = new Conversion()
b: true
s: Conversion@9cab16
h: 9cab16
z = false
b: false
s: false
h: 4d5
*///:~
```

Oznaczone jako komentarze wiersze to konwersje niepoprawne z uwagi na typ zmiennej; ich uruchomienie spowodowałoby zgłoszenie wyjątków.

Zauważ, że konwersja 'b' działa dla każdej testowanej zmiennej, ale choć jest dozwolona dla wszelkich typów argumentów, wcale nie musi się zachowywać zgodnie z oczekiwaniami programisty. Dla wartości podstawowego typu `boolean` i obiektów `Boolean`

wynik konwersji to napis `true` albo `false`. Jednak dla dowolnych innych argumentów, jeśli nie są one wartościami pustymi, wynikiem zawsze jest `true`. Nawet liczbowa wartość zero, która w wielu językach programowania (choćby w C) jest synonimem `false`, da w wyniku konwersji `true` i warto o tym pamiętać.

Oczywiście to nie wszystkie rodzaje konwersji i nie wszystkie opcje specyfikatorów formatu; więcej możesz przeczytać w dokumentacji JDK dla klasy `Formatter`.

Ćwiczenie 5. Dla każdego z podstawowych typów konwersji z powyższej tabeli napisz możliwie najbardziej złożone wyrażenie specyfikatora formatu. Użyj w nim wszystkich możliwych elementów dopuszczalnych dla danego rodzaju konwersji (5).

Metoda String.format()

Java SE5 wzięła też przykład z funkcji `sprintf()` języka C, która służy do tworzenia ciągów znaków. Metoda `String.format()` jest metodą statyczną, przyjmującą argumenty identyczne jak metoda `format()` klasy `Formatter`, ale w przeciwieństwie do niej nie wypisuje utworzonego ciągu na wyjście (ani do wskazanego obiektu strumienia), a tworzy nowy obiekt `String`. Przydaje się to bardzo, jeśli trzeba wywołać `format()` tylko raz:

```
//: strings/DatabaseException.java

public class DatabaseException extends Exception {
  public DatabaseException(int transactionID, int queryID,
    String message) {
    super(String.format("(t%d, q%d) %s", transactionID,
        queryID, message));
  }
  public static void main(String[] args) {
    try {
      throw new DatabaseException(3, 7, "Błąd zapisu");
    } catch(Exception e) {
      System.out.println(e);
    }
  }
} /* Output:
DatabaseException: (t3, q7) Błąd zapisu
*///:~
```

Za kulisami działanie metody `String.format()` sprowadza się do utworzenia egzemplarza `Formatter` i przekazania do niego otrzymanych argumentów; to tylko prosta delegacja, ale jakże poręczna i czytelna.

Narzędzie podglądu szesnastkowego

W ramach drugiego przykładu sporządzimy sobie narzędzie, które pozwoli na podejrzenie w pliku binarnym bajtów wyrażonych jako wartości szesnastkowe. Program wypisuje na wyjście zawartość tablicy bajtów w (jako tako) czytelnym dla człowieka zapisie szesnastkowym, a wykorzystuje do tego metodę `String.format()`:

```
//: net/mindview/util/Hex.java
package net.mindview.util;
import java.io.*;
```

```
public class Hex {
  public static String format(byte[] data) {
    StringBuilder result = new StringBuilder();
    int n = 0;
    for(byte b : data) {
      if(n % 16 == 0)
        result.append(String.format("%05X: ", n));
      result.append(String.format("%02X ", b));
      n++;
      if(n % 16 == 0) result.append("\n");
    }
    result.append("\n");
    return result.toString();
  }
  public static void main(String[] args) throws Exception {
    if(args.length == 0)
      // Test - wyświetlenie pliku skompilowanej klasy:
      System.out.println(
        format(BinaryFile.read("Hex.class")));
    else
      System.out.println(
        format(BinaryFile.read(new File(args[0]))));
  }
} /* Output: (Sample)
00000: CA FE BA BE 00 00 00 31 00 52 0A 00 05 00 22 07
00010: 00 23 0A 00 02 00 22 08 00 24 07 00 25 0A 00 26
00020: 00 27 0A 00 28 00 29 0A 00 02 00 2A 08 00 2B 0A
00030: 00 2C 00 2D 08 00 2E 0A 00 02 00 2F 09 00 30 00
00040: 31 08 00 32 0A 00 33 00 34 0A 00 15 00 35 0A 00
00050: 36 00 37 07 00 38 0A 00 12 00 39 0A 00 33 00 3A
...
*///:~
```

Do otwierania i wczytywania pliku binarnego wykorzystujemy tu inne narzędzie, prezentowane w rozdziale „Wejście-wyjście", a mianowicie klasę `net.mindview.util.BinaryFile`. Metoda `read()` tej klasy zwraca całą zawartość pliku w formie tablicy bajtów.

Ćwiczenie 6. Utwórz klasę zawierającą pola typu `int`, `long`, `float` i `double`. Napisz dla tej klasy metodę `toString()`, która użyje metody `String.format()`, i zaprezentuj prawidłowe działanie klasy (2).

Wyrażenia regularne

Wyrażenia regularne stanowią integralną część standardowych narzędzi systemu Unix, takich jak *sed* oraz *awk* i języków programowania, takich jak Python i Perl (niektórzy mogliby twierdzić, że stanowią one główną przyczynę sukcesu języka Perl). Narzędziami do manipulowania ciągami znaków były wcześniej klasy `String`, `StringBuffer` oraz `StringTokenizer`, jednak w porównaniu do wyrażeń regularnych narzędzia te trzeba uznać za mocno uproszczone.

Wyrażenia regularne są efektywnymi i elastycznymi narzędziami przetwarzania tekstu. Pozwalają na programowe definiowanie skomplikowanych wzorców tekstu, które mają być wykrywane w ciągu wejściowym. Po odkryciu takiego wzorca można w wybrany

sposób zareagować na fakt dopasowania. Choć składnia wyrażeń regularnych z początku onieśmiela, stanowią one zwarty i dynamiczny język programowania, nadający się do rozwiązywania wszelkich zadań przetwarzania tekstu, dopasowywania ciągów, ich edycji i weryfikacji — a wszystko w maksymalnie ogólny sposób.

Podstawy

Wyrażenia regularne to sposób opisu ciągów znaków przy wykorzystaniu ogólnych pojęć, dzięki któremu można powiedzieć: „Jeśli ciąg znaków zawiera podane elementy, to spełnia moje wymagania". Na przykład, aby stwierdzić, że ciąg może, lecz nie musi, być poprzedzony znakiem minusa, należy użyć wyrażenia składającego się ze znaku minus i pytajnika:

```
-?
```

Liczbę całkowitą można opisać jako jedną lub więcej cyfr. W wyrażeniach regularnych cyfra jest reprezentowana przez sekwencję znaków \d. Jeśli posiadasz doświadczenie w wykorzystaniu wyrażeń regularnych w innych językach programowania, od razu zauważysz różnice w sposobie traktowania znaków odwrotnego ukośnika. W innych językach kombinacja \\ oznacza: „Chcę umieścić w wyrażeniu regularnym zwyczajny znak odwrotnego ukośnika". Ta sama kombinacja znaków oznacza w Javie zupełnie coś innego: „Chcę umieścić w wyrażeniu regularnym specjalny znak odwrotnego ukośnika, aby znak umieszczony za nim był traktowany w szczególny sposób". Na przykład, jeśli chcesz oznaczyć jeden lub więcej znaków tworzących słowa, ciąg definiujący wyrażenie regularne powinien przybrać następującą postać: \\w+. Aby umieścić w wyrażeniu regularnym zwyczajny znak odwrotnego ukośnika, należy użyć wyrażenia: \\\\. Stosując znaki nowego wiersza lub tabulacji, wystarczy umieszczać w wyrażeniu regularnym pojedyncze odwrotne ukośniki: \n\t.

W celu zaznaczenia, że podany wcześniej fragment wyrażenia może wystąpić „raz lub więcej razy", należy za nim umieścić znak +. A zatem, aby wyrazić „jedną lub kilka cyfr, które mogą być poprzedzone znakiem minusa", należy użyć wyrażenia:

```
-?\\d+
```

Najprostszym sposobem użycia wyrażeń regularnych jest skorzystanie z wbudowanych możliwości klasy `String`. Na przykład poniższy program sprawdza, czy ciąg zawarty w obiekcie `String` pasuje do powyższego wyrażenia regularnego:

```
//: strings/IntegerMatch.java

public class IntegerMatch {
  public static void main(String[] args) {
    System.out.println("-1234".matches("-?\\d+"));
    System.out.println("5678".matches("-?\\d+"));
    System.out.println("+911".matches("-?\\d+"));
    System.out.println("+911".matches("(-|\\+)?\\d+"));
  }
} /* Output:
true
true
false
true
*///:~
```

Pierwsze dwa ciągi pasują do wyrażenia, ale trzeci zaczyna się od znaku '+', a więc może i reprezentuje liczbę, ale na pewno nieujemną, a więc nie pasuje do wyrażenia regularnego. Gdybyśmy chcieli dopasować liczby dodatnie i ujemne, powinniśmy wyrazić warunek: „ma zaczynać się od + albo –". W wyrażeniach regularnych podwyrażenia grupuje się w nawiasach, a znak pionowej kreski ('|') oznacza alternatywę (OR). Stąd:

```
(-|\\+)
```

oznacza, że w danej części ciągu spodziewamy się znaków '-' lub '+' albo zupełnie niczego (to z racji obecności '?'). Ponieważ sam znak '+' ma w wyrażeniach regularnych znaczenie specjalne, musi zostać poprzedzony ukośnikami, aby pojawił się w nim jako zwykły znak traktowany literalnie.

Jednym z przydatniejszych narzędzi opartych na wyrażeniach regularnych, a wbudowanych w klasę `String`, jest metoda `split()`, która realizuje polecenie „podziel ciąg w miejscach dopasowania podanego wyrażenia regularnego":

```
//: strings/Splitting.java
import java.util.*;

public class Splitting {
  public static String knights =
    "A gdy już znajdziecie żywopłot, musicie " +
    "ściąć najpotężniejsze drzewo w tym lesie... za pomocą... " +
    "śledzia!";
  public static void split(String regex) {
    System.out.println(
      Arrays.toString(knights.split(regex)));
  }
  public static void main(String[] args) {
    split(" "); // Nie zawiera symboli wyrażenia regularnego
    split("\\W+"); // Dopasowuje znaki spoza zbioru
                   // znaków dopuszczalnych w identyfikatorach
                   // (a-z, A-Z, 0-9 i _)
    split("e\\W+"); // Dopasowuje 'e', a za nim znaki
                    // niedozwolone w identyfikatorach
  }
} /* Output:
[A, gdy, już, znajdziecie, żywopłot,, musicie, ściąć, najpotężniejsze, drzewo, w, tym, lesie..., za, pomocą..., śledzia!]
[A, gdy, ju, znajdziecie, ywop, ot, musicie, ci, najpot, niejsze, drzewo, w, tym, lesie, za, pomoc, ledzia]
[A gdy już znajdzieci, ywopłot, musici, ciąć najpotężniejsz, drzewo w tym lesi, za pomocą... śledzia!]
*///:~
```

Przede wszystkim zauważ, że w wyrażeniach regularnych można umieszczać najzwyklejsze znaki — wyrażenie regularne nie musi zawierać znaków specjalnych; widać to w pierwszym wywołaniu `split()`, gdzie rolę całego wyrażenia podziału pełni samotny znak spacji.

Drugie i trzecie wywołanie `split()` operuje w wyrażeniu regularnym symbolem `\W`, które oznacza znak spoza zbioru liter i cyfr (w wersji '`\w`' symbol ten oznacza znak litery albo cyfry). Na wyjściu widać, że z wyodrębnianych podciągów zniknęły znaki interpunkcyjne[3]. W trzecim wywołaniu miejsce podziału podciągów zostało określone jako miejsce występowania litery 'e' uzupełnionej znakiem spoza zbioru liter i cyfr.

[3] Widać też, że klasa „liter i cyfr" wyrażona jako symbol `\w` nie obejmuje polskich liter, co zaburza podział w miejscach ich występowania — *przyp. tłum.*

Przeciążona wersja `String.split()` pozwala na ograniczanie liczby podziałów.

Ostatnim narzędziem klasy `String` operującym na wyrażeniach regularnych jest mechanizm zastępowania podciągów. Można za jego pomocą zastąpić pierwsze wystąpienie albo wszystkie wystąpienia podciągu określonego wyrażeniem regularnym:

```
//: strings/Replacing.java
import static net.mindview.util.Print.*;

public class Replacing {
  static String s = Splitting.knights;
  public static void main(String[] args) {
    print(s.replaceFirst("z\\w+", "zasadzicie"));
    print(s.replaceAll("żywopłot|drzewo|śledzi","pomidor"));
  }
} /* Output:
A gdy już zasadzicie żywopłot, musicie ściąć najpotężniejsze drzewo w tym lesie... za pomocą... śledzia!
A gdy już znajdziecie pomidor, musicie ściąć najpotężniejsze pomidor w tym lesie... za pomocą... pomidora!
*///:~
```

Pierwsze wyrażenie dopasowuje literę 'z', a za nią nieokreślonej długości znaków liter i cyfr (zauważ, że tym razem w '`\w`' mamy małe '`w`'). Zastępowanie ma dotyczyć jedynie pierwszego dopasowania, a więc słowa „znajdziecie", które jest zamieniane na „zasadzicie".

Drugie wyrażenie dopasowuje dowolne z trzech słów rozdzielonych znakiem alternatywy i zastępuje dopasowanie podciągiem „pomidor". Zastępowanie dotyczy wszystkich dopasowań.

Wkrótce się przekonasz, że znacznie potężniejsze narzędzia wyrażeń regularnych znajdują się poza klasą `String` — tam będzie można przeprowadzić zastępowanie za pomocą wywołań metod. Wyrażenia regularne spoza klasy `String` są też znacznie wydajniejsze, co ma znaczenie zwłaszcza tam, gdzie mają być intensywnie wykorzystywane.

Ćwiczenie 7. Na podstawie dokumentacji klasy `java.util.regex.Pattern` napisz i przetestuj wyrażenie regularne, które sprawdzi, czy zdanie zaczyna się wielką literą i kończy kropką (5).

Ćwiczenie 8. Podziel ciąg `Splitting.knights`; elementami podziału mają być słowa „w" i „za" (2).

Ćwiczenie 9. Na podstawie dokumentacji klasy `java.util.regex.Pattern` zastąp wszystkie samogłoski w ciągu `Splitting.knights` znakami podkreślenia (4).

Tworzenie wyrażeń regularnych

Naukę wyrażeń regularnych można zacząć od poznania wybranych, najbardziej przydatnych konstrukcji. Pełną listę konstrukcji wykorzystywanych przy tworzeniu wyrażeń regularnych można znaleźć w dokumentacji JDK dla klasy `Pattern`, należącej do pakietu `java.util.regex`.

Znaki	
B	Określona litera, w tym przypadku B.
\xhh	Znak o wartości szesnastkowej 0xhh.
\uhhhh	Znak Unicode reprezentowany przez liczbę szesnastkową 0xhhhh.
\t	Znak tabulacji.
\n	Znak nowego wiersza.
\r	Znak powrotu karetki.
\f	Znak przesunięcia strony.
\e	Znak escape.

Ogromna potęga wyrażeń regularnych uwidacznia się dopiero w momencie, gdy zaczniemy definiować klasy znaków. Poniżej przedstawiłem typowe sposoby definiowania klas znaków oraz niektóre z klas predefiniowanych:

Klasy znaków	
.	Reprezentuje dowolny znak.
[abc]	Dowolny ze znaków a, b lub c (to samo co zapis a|b|c).
[^abc]	Dowolny znak oprócz a, b lub c (negacja).
[a-zA-Z]	Dowolny znak z zakresu od a do z lub od A do Z (zakres).
[abc[hij]]	Dowolny ze znaków a, b, c, h, i lub j (to samo co zapis a|b|c|h|i|j) (suma).
[a-z&&[hij]]	Litera h, i lub j (część wspólna klas).
\s	Odstępy (znaki spacji, tabulacji, nowego wiersza, przesunięcia strony i powrotu karetki).
\S	Dowolny znak z wyjątkiem jednego z odstępów ([^\s]).
\d	Cyfra — [0-9].
\D	Dowolny znak z wyjątkiem cyfry — [^0-9].
\w	Dowolny ze znaków tworzących słowa — [a-zA-Z_0-9].
\W	Dowolny znak oprócz znaków tworzących słowa — [^\w].

Przedstawiam tutaj wyłącznie przykłady; bez wątpienia będziesz chciał zapisać adres strony zawierającej dokumentację klasy `java.util.regex.Pattern` na liście ulubionych bądź w menu *Start*, aby mieć łatwy dostęp do wszelkich możliwych wzorców wyrażeń regularnych.

Operatory logiczne	
XY	Znak X, a po nim znak Y.
X|Y	X lub Y.
(X)	*Pobieranie grupy*. W dalszej części wyrażenia można się odwołać do *i*-tej pobranej grupy, używając w tym celu zapisu \i.

Znaki kotwiczące dopasowanie	
^	Początek wiersza.
$	Koniec wiersza.
\b	Granica słowa.
\B	Przeciwieństwo granicy słowa.
\G	Koniec poprzedniego dopasowania.

Każdy z poniższych ciągów stanowi przykład poprawnego wyrażenia regularnego, umożliwiającego odszukanie sekwencji liter „Rudolph":

```
//: strings/Rudolph.java

public class Rudolph {
  public static void main(String[] args) {
    for(String pattern : new String[]{ "Rudolph",
      "[rR]udolph", "[rR][aeiou][a-z]ol.*", "R.*" })
      System.out.println("Rudolph".matches(pattern));
  }
} /* Output:
true
true
true
true
*///:~
```

Naszym celem nie powinno być, rzecz jasna, tworzenie możliwie zawiłych wyrażeń regularnych — chodzi raczej o wyrażenia minimalistyczne, jak najprostsze, niezbędne do wykonania zadania. Przekonasz się zresztą, że kiedy już zaczniesz stosować wyrażenia regularne w swoich programach, będziesz wykorzystywał własny kod w roli ściągawki przy tworzeniu następnych wyrażeń. Warto więc dbać o ich czytelność.

Kwantyfikatory

Kwantyfikatory określają sposób, w jaki wzorzec jest dopasowywany do tekstu wejściowego:

- *Zachłanny*: kwantyfikatory domyślnie są „zachłanne", chyba że działanie to zostanie jawnie zmienione w wyrażeniu. Wyrażenie zachłanne odnajduje wszystkie możliwe fragmenty ciągu pasujące do wzorca. Często popełnianym błędem jest założenie, że wzorzec zostanie dopasowany tylko do pierwszej pasującej grupy znaków, podczas gdy w rzeczywistości jest on zachłanny i obejmie możliwie największą liczbę znaków pasujących do niego.
- *Niechętny*: ten kwantyfikator powoduje, że do wzorca zostanie dopasowana minimalna niezbędna liczba znaków; jest on reprezentowany za pomocą znaku zapytania. Jest on także określany jako: *leniwy*, *minimalnie dopasowujący* lub *nie-chciwy*.
- *Własnościowy*: aktualnie dostępny wyłącznie w języku Java. Jest to bardziej zaawansowany kwantyfikator, którego zapewne nie będziesz stosować od razu. W przypadku dopasowania wyrażenia regularnego do ciągu znaków generowanych jest wiele stanów, aby w razie niepowodzenia można było powrócić do stanu początkowego. Kwantyfikatory własnościowe nie zachowują tych stanów pośrednich,

więc powrót do stanu początkowego nie jest możliwy. Można ich używać, aby uniemożliwić wymknięcie się wyrażenia regularnego spod kontroli, jak również w celu poprawienia efektywności działania wyrażeń regularnych.

Zachłanny	Niechętny	Własnościowy	Odpowiada
`X?`	`X??`	`X?+`	Wyrażeniu `X`, które może wystąpić raz lub nie wystąpić w ogóle.
`X*`	`X*?`	`X*+`	Wyrażeniu `X`, które może nie wystąpić w ogóle lub wystąpić dowolną ilość razy.
`X+`	`X+?`	`X++`	Wyrażeniu `X`, które może wystąpić raz lub większą liczbę razy.
`X{n}`	`X{n}?`	`X{n}+`	`n` powtórzeniom wyrażenia `X`.
`X{n,}`	`X{n,}?`	`X{n,}+`	Co najmniej `n` wystąpieniom wyrażenia `X`.
`X{n,m}`	`X{n,m}?`	`X{n,m}+`	Co najmniej `n` wystąpieniom wyrażenia `X`, które jednak nie powtarza się więcej niż `m` razy.

Koniecznie należy pamiętać, że w niektórych sytuacjach zapewnienie poprawnego działania wyrażenia `X`, będzie wymagać zapisania go w nawiasach. Na przykład:

```
abc+
```

Mogłoby się wydawać, że powyższe wyrażenie odpowiada jednemu lub kilku wystąpieniom ciągu `abc`, a przetwarzając za jego pomocą ciąg znaków `abcabcabc`, można by uzyskać trzy pasujące fragmenty. Jednak w rzeczywistości powyższe wyrażenie oznacza: „Odszukaj ciąg `ab`, po którym ma się znajdować co najmniej jedna litera `c`". Aby odszukać jedno lub kilka powtórzeń ciągu `abc`, należy użyć wyrażenia:

```
(abc)+
```

Łatwo można się pomylić, tworząc wyrażenia regularne — to zupełnie nowy język dodany do Javy.

CharSequence

Interfejs o nazwie `CharSequence` ustanawia uogólnioną definicję sekwencji znaków jako tworu bardziej abstrakcyjnego niż ciągi znaków, czyli klasy `String` i `StringBuilder` czy `StringBuffer`:

```
interface CharSequence {
  charAt(int i);
  length();
  subSequence(int start, int end);
  toString();
}
```

Interfejs ten implementują wszystkie klasy wymienione w poprzednim akapicie. Wiele operacji wykonywanych na wyrażeniach regularnych pobiera argumenty `CharSequence`.

Klasy Pattern oraz Matcher

Zwykle zamiast stosować narzędzia wyrażeń regularnych wbudowane w klasę `String` kompiluje się wyrażenia regularne do postaci samodzielnych obiektów. Trzeba w tym celu zaimportować pakiet `java.util.regex`, a potem skompilować wyrażenie regularne za pomocą statycznej metody `Pattern.compile()`. W ten sposób na bazie argumentu `String` powstanie obiekt klasy `Pattern`. Obiektu tego można używać, wywołując jego metodę `matcher()`, wytwarzającą z kolei obiekt klasy `Matcher` z bogatym zestawem operacji (są one wymienione w dokumentacji JDK dla klasy `java.util.regex.Matcher`). Na przykład metoda `replaceAll()` zastępuje ciągiem podanym w wywołaniu wszystkie dopasowania wyrażenia regularnego.

Pierwszy przykład zastosowania wyrażeń regularnych pozwala na sprawdzenie wejściowego ciągu znaków przy użyciu podanego wyrażenia. Pierwszym argumentem wywołania programu jest sprawdzany ciąg znaków, a kolejnymi wyrażenia regularne wykorzystywane do przeprowadzenia testu. W systemach Unix i Linux, podając wyrażenia regularne w wierszu wywołania programu, należy je zapisać w cudzysłowach. Program może być przydatny podczas testowania wyrażeń regularnych i sprawdzania, czy zachowują się one w spodziewany sposób.

```
// : strings/TestRegularExpression.java
// Program pozwalający na łatwe testowanie wyrażeń regularnych.
// {Args: abcabcabcdefabc "abc+" "(abc)+" "(abc){2,}" }
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class TestRegularExpression {
  public static void main(String[] args) {
    if(args.length < 2) {
      print("Stosowanie:\njava TestRegularExpression " +
        "ciagznaków wyrażenieregularne+");
      System.exit(0);
    }
    print("Wejście: \"" + args[0] + "\"");
    for(String arg : args) {
      print("Wyrażenie regularne: \"" + arg + "\"");
      Pattern p = Pattern.compile(arg);
      Matcher m = p.matcher(args[0]);
      while(m.find()) {
        print("Dopasowanie \"" + m.group() + "\" na pozycjach " +
          m.start() + "-" + (m.end() - 1));
      }
    }
  }
} /* Output:
Wejście: "abcabcabcdefabc"
Wyrażenie regularne: "abcabcabcdefabc"
Dopasowanie "abcabcabcdefabc" na pozycjach 0-14
Wyrażenie regularne: "abc+"
Dopasowanie "abc" na pozycjach 0-2
Dopasowanie "abc" na pozycjach 3-5
Dopasowanie "abc" na pozycjach 6-8
Dopasowanie "abc" na pozycjach 12-14
Wyrażenie regularne: "(abc)+"
Dopasowanie "abcabcabc" na pozycjach 0-8
```

```
Dopasowanie "abc" na pozycjach 12-14
Wyrażenie regularne: "(abc){2,}"
Dopasowanie "abcabcabc" na pozycjach 0-8
*///:~
```

Obiekt `Pattern` reprezentuje skompilowaną wersję wyrażenia regularnego. Zgodnie z tym, co pokazałem w powyższym przykładzie, posługując się metodą `matcher()` obiektu `Pattern` oraz wejściowym ciągiem znaków, można uzyskać obiekt `Matcher`. Klasa `Pattern` posiada też metodę statyczną `matches()`:

```
static boolean matches(String wyrażenieRegularne, CharSequence ciągWejściowy)
```

która pozwala sprawdzić, czy podane `wyrażenieRegularne` występuje w `ciąguWejściowym`, oraz metodę `split()`, zwracającą tablicę znaków powstałą w wyniku podzielenia `ciąguWejściowego` w miejscach, w których znaleziono fragmenty pasujące do `wyrażeniaRegularnego`.

Obiekt `Matcher` generowany jest przez wywołanie metody `Pattern.matcher()` z wejściowym ciągiem znaków. Zapewnia on dostęp do wyników, a jego metody pozwalają sprawdzić, czy możliwe są różne rodzaje dopasowania wyrażenia do ciągu wejściowego:

```
boolean matches()
boolean lookingAt()
boolean find()
boolean find(int start)
```

Metoda `matches()` zwraca wartość `true`, jeśli wzorzec pasuje do całego wejściowego ciągu znaków; natomiast metoda `lookingAt()` — gdy wzorzec pasuje do ciągu wejściowego i zaczyna się na samym jego początku.

Ćwiczenie 10. Określ, czy przedstawione poniżej wyrażenia regularne można dopasować do ciągu znaków „Java już obsługuje wyrażenia regularne" (2):

```
^Java
\Breg.*
u.e\s+w(y|i)r
s?
s*
s+
s{4}
s{1,}.
s{0,3}
```

Ćwiczenie 11. Użyj wyrażenia:

```
(?i)((^[aeiou])|(\s+[aeiou]))\w+?[aeiou]\b
```

do sprawdzenia ciągu znaków (2):

```
Agata zjadła osiem ananasów i ostrygę, a Anita obyła się smakiem
```

Metoda find()

Metoda `Matcher.find()` umożliwia odnalezienie wielu fragmentów sekwencji znaków pasujących do podanego wyrażenia regularnego. Na przykład:

```
//: strings/Finding.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class Finding {
  public static void main(String[] args) {
    Matcher m = Pattern.compile("\\w+")
      .matcher("Wieczorne akrobacje mrocznych nietoperzy");
    while(m.find())
      printnb(m.group() + " ");
    print();
    int i = 0;
    while(m.find(i)) {
      printnb(m.group() + " ");
      i++;
    }
  }
} /* Output:
Wieczorne akrobacje mrocznych nietoperzy
Wieczorne ieczorne eczorne czorne zorne orne rne ne e akrobacje akrobacje krobacje robacje obacje bacje
acje cje je e mrocznych mrocznych rocznych ocznych cznych znych nych ych ch h nietoperzy nietoperzy
ietoperzy etoperzy toperzy operzy perzy erzy rzy zy y
*///:~
```

Wzorzec `\\w+` podzieli wejściowy ciąg znaków na poszczególne słowa. Metoda `find()` przypomina raczej iterator przesuwający się po wejściowym ciągu znaków. Z kolei druga wersja metody `find()` umożliwia podanie liczby całkowitej, określającej położenie znaku, od którego należy rozpocząć poszukiwanie; jej wywołanie powoduje określenie pozycji początkowej wyszukiwania na podstawie przekazanego argumentu, o czym można się przekonać na podstawie wyników działania programu.

Grupy

Grupy są fragmentami wyrażenia regularnego, do których można się odwoływać w jego dalszych częściach za pomocą numerów. Grupy oznaczane są w wyrażeniu przy użyciu nawiasów. Grupa zerowa odpowiada całemu wyrażeniu regularnemu, grupa pierwsza — pierwszemu fragmentowi wyrażenia zapisanemu w nawiasach itd. A zatem w poniższym wyrażeniu:

```
A(B(C))D
```

są dostępne trzy grupy: grupę zerową stanowi ciąg `ABCD`, pierwszą `BC`, a drugą `C`.

Obiekt `Matcher` udostępnia metody pozwalające na uzyskanie informacji o grupach. Są nimi:

Metoda	Działanie
`public int groupCount()`	Zwraca liczbę grup odnalezionych przy dopasowywaniu wzorca. Zwracana wartość nie uwzględnia grupy zerowej.
`public String group()`	Zwraca zawartość grupy zerowej (cały dopasowany ciąg) wyznaczonej podczas ostatniej operacji dopasowywania (na przykład wywołania metody `find()`).
`public String group(int i)`	Zwraca grupę o określonym numerze wyznaczoną podczas ostatniej operacji dopasowywania. Jeśli operacja zakończyła się sukcesem, lecz określona grupa nie została dopasowana do żadnego fragmentu wejściowego ciągu znaków, zwracana jest wartość `null`.

Metoda	Działanie
`public int start(int grupa)`	Zwraca początkowy indeks położenia grupy wyznaczonej podczas ostatniej operacji dopasowywania.
`public int end(int grupa)`	Zwraca, powiększony o jeden, indeks ostatniego znaku grupy wyznaczonej podczas ostatniej operacji dopasowywania.

Poniżej przedstawiłem przykład:

```
//: strings/Groups.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class Groups {
  static public final String POEM =
    "Brzdeśniało już; ślimonne prztowie\n" +
    "Wyrło i warło się w gulbieży:\n" +
    "Zmimszałe ćwiły borogowie\n" +
    "I rcie grdypały z mrzerzy.\n" +
    "O, strzeż się, synu, Dziaberlaka!\n" +
    "Łap pazurzastych, zębnej paszczy!\n" +
    "Omiń Dziupdziupa, złego ptaka.\n" +
    "Z którym się Brutrwiel piastrzy!\n";
  public static void main(String[] args) {
    Matcher m =
      Pattern.compile("(?m)(\\S+)\\s+((\\S+)\\s+(\\S+))$")
        .matcher(POEM);
    while(m.find()) {
      for(int j = 0; j <= m.groupCount(); j++)
        printnb("[" + m.group(j) + "]");
      print();
    }
  }
} /* Output:
[już; ślimonne prztowie][już;][ślimonne prztowie][ślimonne][prztowie]
[się w gulbieży;][się][w gulbieży;][w][gulbieży;]
[Zmimszałe ćwiły borogowie][Zmimszałe][ćwiły borogowie][ćwiły][borogowie]
[grdypały z mrzerzy.][grdypały][z mrzerzy.][z][mrzerzy.]
[się, synu, Dziaberlaka!][się,][synu, Dziaberlaka!][synu,][Dziaberlaka!]
[pazurzastych, zębnej paszczy!][pazurzastych,][zębnej paszczy!][zębnej][paszczy!]
[Dziupdziupa, złego ptaka,][Dziupdziupa,][złego ptaka,][złego][ptaka,]
[się Brutrwiel piastrzy!][się][Brutrwiel piastrzy!][Brutrwiel][piastrzy!]
*///:~
```

W programie wykorzystana została pierwsza część wiersza Lewisa Carrolla pt. „Jabberwocky"[4] ze zbioru *Through the Looking Glass*. Jak widać, we wzorcu wyrażenia regularnego zostały umieszczone grupy odpowiadające dowolnej liczbie dowolnych znaków, z wyjątkiem odstępów (\S+), po których ma wystąpić dowolnie wiele znaków odstępu (\s+). Naszym celem jest określenie ostatnich trzech słów każdego wiersza tekstu. Koniec wiersza oznaczany jest symbolem `$`. Jednak standardowy sposób działania polega na dopasowaniu symbolu `$` do samego końca całej wejściowej sekwencji znaków; dlatego też należy jawnie zażądać, by wyrażenie regularne zwracało uwagę na znaki nowego wiersza. Efekt ten można uzyskać, umieszczając na początku wyrażenia regularnego flagę `(?m)` (flagi zostaną przedstawione w dalszej części rozdziału).

[4] W przykładzie użyto tłumaczenia Stanisława Barańczaka, w którym wiersz nosi nazwę *DZIABERLIADA*.

Ćwiczenie 12. Zmień plik *Groups.java* tak, aby zliczyć (bez powtórzeń) wszystkie słowa, które nie zaczynają się od wielkiej litery (5).

Metody start() oraz end()

Po udanej operacji dopasowywania metoda `start()` zwraca indeks początku fragmentu wejściowego ciągu znaków pasującego do wyrażenia regularnego, a metoda `end()` — indeks ostatniej litery tego fragmentu powiększony o jeden. Jeśli ostatnia operacja dopasowywania nie zakończyła się pomyślnie, wywołanie którejkolwiek z tych metod spowoduje zgłoszenie wyjątku `IllegalStateException` (podobnie stanie się, gdy któraś z tych metod zostanie wywołana przed wykonaniem operacji dopasowywania). Kolejny program[5] przestawia wykorzystanie metod `matches()` oraz `lookingAt()`:

```
//: strings/StartEnd.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class StartEnd {
  public static String input =
    "Jak długo będzie istniała niesprawiedliwość.\n" +
    "kiedykolwiek zapłacze targathiańskie dziecko.\n" +
    "gdziekolwiek pośród gwiazd\n zabrzmi wezwanie pomocy...\n" +
    "Będziemy tam\n." +
    "Ten wspaniały statek i ta wspaniała załoga...\n" +
    "Nigdy nie zrezygnuje! Nigdy się nie podda!";
  private static class Display {
    private boolean regexPrinted = false;
    private String regex;
    Display(String regex) { this.regex = regex; }
    void display(String message) {
      if(!regexPrinted) {
        print(regex);
        regexPrinted = true;
      }
      print(message);
    }
  }
  static void examine(String s, String regex) {
    Display d = new Display(regex);
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(s);
    while(m.find())
      d.display("find() '" + m.group() +
        "' start = "+ m.start() + " end = " + m.end());
    if(m.lookingAt()) // Wywołanie reset() zbędne
      d.display("lookingAt() start = "
        + m.start() + " end = " + m.end());
    if(m.matches()) // Wywołanie reset() zbędne
      d.display("matches() start = "
        + m.start() + " end = " + m.end());
  }
  public static void main(String[] args) {
    for(String in : input.split("\n")) {
      print("Wejście : " + in);
```

[5] A w programie płomienna przemowa komandora Taggarta z *Kosmicznej Załogi*.

```
      for(String regex : new String[]{"\\w*zie\\w*",
        "\\w*kolwiek", "T\\w+", "Nigdy.*?!"})
        examine(in, regex);
    }
  }
} /* Output:
Wejście : Jak długo będzie istniała niesprawiedliwość,
\w*zie\w*
find() 'dzie' start = 12 end = 16
Wejście : kiedykolwiek zapłacze targathiańskie dziecko,
\w*zie\w*
find() 'dziecko' start = 37 end = 44
\w*kolwiek
find() 'kiedykolwiek' start = 0 end = 12
lookingAt() start = 0 end = 12
T\w+
find() targathia' start = 22 end = 31
Wejście : gdziekolwiek pośród gwiazd
\w*zie\w*
find() 'gdziekolwiek' start = 0 end = 12
lookingAt() start = 0 end = 12
\w*kolwiek
find() 'gdziekolwiek' start = 0 end = 12
lookingAt() start = 0 end = 12
Wejście :  zabrzmi wezwanie pomocy...
Wejście : Będziemy tam
\w*zie\w*
find() 'dziemy' start = 2 end = 8
Wejście : .Ten wspaniały statek i ta wspaniała załoga...
T\w+
find() 'Ten' start = 1 end = 4
Wejście : Nigdy nie zrezygnuje! Nigdy się nie podda!
Nigdy.*?!
find() 'Nigdy nie zrezygnuje!' start = 0 end = 21
find() 'Nigdy się nie podda!' start = 22 end = 42
lookingAt() start = 0 end = 21
matches() start = 0 end = 42
*///:~
```

Warto zauważyć, że metoda `find()` jest w stanie odnaleźć wyrażenie regularne w dowolnym miejscu wejściowego ciągu znaków, natomiast metoda `lookingAt()` oraz `matches()` zwrócą `true` wyłącznie w przypadkach, gdy sekwencja pasująca do wyrażenia regularnego rozpoczyna się na samym początku wejściowego ciągu znaków. Metoda `matches()` zwraca `true` wyłącznie w przypadku, gdy *cały* ciąg wejściowy pasuje do wyrażenia regularnego, natomiast metoda `lookingAt()`[6] — gdy do wyrażenia pasuje początkowa część ciągu wejściowego.

Ćwiczenie 13. Zmień program *StartEnd.java* tak, aby w roli wejścia używał `Groups.POEM`, ale wciąż generował pozytywne wyniki dla `find()`, `lookingAt()` i `matches()` (2).

[6] Nie mam najmniejszego pojęcia, jak twórcy języka wpadli na pomysł nadania tej metodzie takiej nazwy ani co ma ona oznaczać. Uspokaja jednak pewność, że ktokolwiek wymyślił tę całkowicie nieintuicyjną nazwę, na pewno wciąż jest zatrudniony w firmie Sun i prowadzona przez niego polityka nieprzeglądania projektów kodu wciąż jest stosowana. Przepraszam za sarkazm, lecz po kilku latach takie sytuacje stają się już męczące.

Flagi wzorców

Alternatywna wersja metody `compile()` akceptuje flagi zmieniające sposób dopasowywania wyrażenia regularnego:

```
Pattern Pattern.compile(String wyrażenieRegularne, int flaga)
```

gdzie `flaga` jest jedną ze stałych zdefiniowanych w klasie `Pattern`:

Flaga kompilacji	Działanie
`Pattern.CANON_EQ`	Dwa znaki będą sobie odpowiadały wtedy i tylko wtedy, gdy będą sobie odpowiadały ich pełne rozkłady kanoniczne. Na przykład, jeśli flaga ta zostanie wykorzystana, to wyrażenie `\u003F` będzie pasować do ciągu znaków ?. Domyślny sposób dopasowywania wyrażeń regularnych nie wykorzystuje porównywania pełnych postaci kanonicznych.
`Pattern.CASE_INSENSITIVE` `(?i)`	Domyślnie, dopasowując wyrażenia bez uwzględniania wielkości liter, zakłada się, że porównywane są wyłącznie znaki należące do zbioru US-ASCII. Flaga ta umożliwia dopasowywanie wzorca bez zwracania uwagi na wielkość liter. Aby nie uwzględniać wielkości liter przy porównywaniu wyrażeń regularnych z ciągami zawierającymi dowolne znaki Unicode, oprócz tej flagi należy także użyć flagi `UNICODE_CASE`.
`Pattern.COMMENTS` `(?x)`	W tym trybie pomijane są wszystkie znaki odstępu, pomijane są wszystkie odstępy, a osadzone komentarze rozpoczynające się od znaku # są pomijane aż do końca wiersza. Istnieje także flaga umożliwiająca włączanie trybu obsługi znaków końca wiersza stosowanych w systemie Unix.
`Pattern.DOTALL` `(?s)`	W tym trybie wyrażenie . odpowiada dowolnemu znakowi, w tym także znakom reprezentującym koniec wiersza. Domyślnie wyrażenie to odpowiada wszystkim znakom z wyjątkiem znaków końca wiersza.
`Pattern.MULITLINE` `(?m)`	W trybie wielowierszowym wyrażenia ^ oraz $ odpowiada odpowiednio początkowi i końcowi wiersza. Pierwsze z tych wyrażeń odpowiada także początkowi całego wejściowego ciągu znaków, a drugie — końcowi tego ciągu. Domyślnie wyrażenie ^ odpowiada wyłącznie początkowi całego wejściowego ciągu, a wyrażenie $ — wyłącznie końcowi tego ciągu.
`Pattern.UNICODE_CASE` `(?u)`	W przypadku użycia tej flagi oraz flagi `CASE_INSENSITIVE` podczas dopasowywania wyrażeń regularnych nie jest uwzględniana wielkość liter, a sposób porównywania ciągów jest zgodny ze standardem Unicode. Domyślnie, jeśli wielkość liter jest ignorowana, zakłada się, że porównywane są wyłącznie znaki należące do zbioru US-ASCII.
`Pattern.UNIX_LINES` `(?d)`	W tym trybie jedynie znak \n jest rozpoznawany w działaniu wyrażeń ., ^ oraz $.

Spośród przedstawionych flag najbardziej przydatne są `Pattern.CASE_INSENSITIVE`, `Pattern. MULTILINE` oraz `Pattern.COMMENTS` (umożliwiający zachowanie przejrzystości i stosowany przy tworzeniu dokumentacji). Należy zauważyć, że sposób działania cha-

rakterystyczny dla większości z tych flag można także uzyskać, umieszczając w wyrażeniu regularnym odpowiednią kombinację znaków, przedstawioną w tabeli poniżej flag. Kombinację tę należy umieszczać przed wybranym miejscem wyrażenia, od którego chcemy zmodyfikować sposób dopasowywania.

Efekty działania flag, zarówno tych przedstawionych w powyższej tabeli, jak i wszystkich pozostałych, można łączyć przy użyciu operatora OR (|):

```
//: strings/ReFlags.java
import java.util.regex.*;

public class ReFlags {
  public static void main(String[] args) {
    Pattern p =  Pattern.compile("^java",
      Pattern.CASE_INSENSITIVE | Pattern.MULTILINE);
    Matcher m = p.matcher(
      "java ma regex\nJava ma regex\n" +
      "JAVA ma całkiem niezłe wyrażenia regularne\n" +
      "Wyrażenia regularne są już w języku Java");
    while(m.find())
      System.out.println(m.group());
  }
} /* Output:
java
Java
JAVA
*///:~
```

Wyrażenie regularne użyte w powyższym programie odnajduje wiersze rozpoczynające się od słów: „java", „Java", „JAVA" itd., starając się odnaleźć je w każdym wierszu wielowierszowego ciągu znaków (proces dopasowywania rozpoczyna się na początku sekwencji znaków oraz od każdego znaku nowego wiersza w tej sekwencji). Zauważ, że metoda `group()` zwraca wyłącznie odszukane fragmenty ciągu wejściowego.

metoda split()

Operacja podziału dzieli wejściowy ciąg znaków w miejscach wystąpienia wyrażenia regularnego i generuje tablicę obiektów `String`.

```
String[] split(CharSequence sekwencjaZnaków)
String[] split(CharSequence sekwencjaZnaków, int limit)
```

Umożliwia ona łatwe i wygodne podzielenie wejściowego ciągu znaków w miejscach występowania pewnej wspólnej granicy:

```
//: strings/SplitDemo.java
import java.util.regex.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class SplitDemo {
  public static void main(String[] args) {
    String input =
      "To!!niezwykłe zastosowanie!!znaków!!wykrzyknika";
    print(Arrays.toString(
```

```
      Pattern.compile("!!").split(input)));
    // Tylko dla pierwszych trzech:
    print(Arrays.toString(
      Pattern.compile("!!").split(input, 3)));
  }
} /* Output:
[To, niezwykłe zastosowanie, znaków, wykrzyknika]
[To, niezwykłe zastosowanie, znaków!!wykrzyknika]
*///:~
```

Druga wersja metody `split()` ogranicza liczbę fragmentów, na które będzie dzielony wejściowy ciąg znaków.

Ćwiczenie 14. Przepisz program *SplitDemo.java* z użyciem metody `String.split()` (1).

Operacje zastępowania

Wyrażenia regularne stają się szczególnie przydatne do zastępowania tekstów. Oto dostępne metody:

Metoda	Działanie
`replaceFirst(String ciągZastępujący)`	Zastępuje pierwszy fragment ciągu wejściowego pasujący do wyrażenia regularnego ciągiem podanym w wywołaniu metody.
`replaceAll(String ciągZastępujący)`	Zastępuje wszystkie fragmenty ciągu wejściowego pasujące do wyrażenia regularnego ciągiem podanym w wywołaniu metody.
`appendReplacement (StringBuffer sBufor, String ciągZastępujący)`	Metoda ta nie zastępuje pierwszego lub wszystkich pasujących fragmentów, jak to czynią odpowiednio metody `replaceFirst()` oraz `replaceAll()`, lecz zamiast tego przeprowadza operację zastępowania etapami. Metoda ta jest *niezwykle ważna*, gdyż umożliwia wywoływanie innych metod oraz realizację innych operacji w celu zmiany `ciąguZastępującego` (w przypadku korzystania z metody `replaceFirst()` oraz `replaceAll()` ciąg ten jest niezmienny). Wykorzystując tę metodę, można rozdzielać poszczególne grupy wyrażenia i tworzyć narzędzia zastępujące o ogromnych możliwościach.
`appendTail(StringBuffer sBufor, String ciągZastępujący)`	Metoda ta jest stosowana po jednym lub kilku wywołaniach metody `appendReplacement()` w celu skopiowania pozostałej części ciągu wejściowego.

Poniższy przykład przedstawia zastosowanie wszystkich operacji zastępowania. Program wczytuje komentarz blokowy umieszczony na początku jego kodu i wykorzystuje go jako ciąg wejściowy podczas realizacji kolejnych operacji:

```
//: strings/TheReplacements.java
import java.util.regex.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

/*! Oto blok tekstu, który zostanie wykorzystany
  jako ciąg wejściowy w operacjach na wyrażeniach
  regularnych. Program w pierwszej kolejności
  pobierze zawartość bloku, poszukując specjalnych
  ograniczników, a następnie wykorzysta ją
  w dalszych operacjach !*/
```

```
public class TheReplacements {
  public static void main(String[] args) throws Exception {
    String s = TextFile.read("TheReplacements.java");
    // Dopasowanie powyższego komentarza bloku tekstu:
    Matcher mInput =
      Pattern.compile("/\\*!(.*)!\\*/", Pattern.DOTALL)
        .matcher(s);
    if(mInput.find())
      s = mInput.group(1); // Wyłuskany przez grupę w nawiasach
    // Zastępuje dwa lub więcej odstępów jednym znakiem odstępu:
    s = s.replaceAll(" {2,}", " ");
    // Usuwa wszelkie ostępy z początku wiersza
    // Koniecznie należy wykorzystać tryb MULTILINE:
    s = s.replaceAll("(?m)^ +", "");
    print(s);
    s = s.replaceFirst("[aeiouyąęó]", "(SAMOGŁOSKI1)");
    StringBuffer sbuf = new StringBuffer();
    Pattern p = Pattern.compile("[aeiouyąęó]");
    Matcher m = p.matcher(s);
    // Przetwarzanie informacji podczas realizacji
    // operacji zastępowania:
    while(m.find())
      m.appendReplacement(sbuf, m.group().toUpperCase());
    // Dodanie pozostałej części ciągu wejściowego:
    m.appendTail(sbuf);
    print(sbuf);
  }
} /* Output:
Oto blok tekstu, który zostanie wykorzystany
jako ciąg wejściowy w operacjach na wyrażeniach
regularnych. Program w pierwszej kolejności
pobierze zawartość bloku, poszukując specjalnych
ograniczników, a następnie wykorzysta ją
w dalszych operacjach
Ot(SAMOGŁOSKI1) blOk tEkstU, ktÓrY zOstAnIE wYkOrzYstAnY
jAkO cIĄg wEjŚcIOwY w OpErAcjAch nA wYrAŻEnIAch
rEgUlArnYch. PrOgrAm w pIErwszEj kOlEjnOścI
pObIErzE zAwArtOść blOkU, pOszUkUjĄc spEcjAlnYch
OgrAnIcznIkÓw, A nAstĘpnIE wYkOrzYstA jĄ
w dAlszYch OpErAcjAch
*///:~
```

Plik jest otwierany i wczytywany przy użyciu klasy `TextFile` z biblioteki `net.mindview.util` (kod programu zostanie zaprezentowany w rozdziale „Wejście-wyjście"). Statyczna metoda `read()` tej klasy wczytuje całość pliku i zwraca jego zawartość w obiekcie `String`. Następnie program tworzy obiekt `mInput`, którego zadaniem jest pobranie całego tekstu umieszczonego pomiędzy kombinacjami znaków `/*!` oraz `!*/` (zwróć uwagę na użycie nawiasów w wyrażeniu regularnym). W uzyskanym ciągu znaków występujące obok siebie dwa znaki odstępu zostają zamienione na jeden, a wszystkie odstępy umieszczone na początku wiersza zostają usunięte (aby było możliwe usunięcie odstępów z początku wszystkich wierszy, a nie wyłącznie samego początku całego ciągu, konieczne jest włączenie trybu wielowierszowego). Obie te operacje są wykonywane przy użyciu metody `replaceAll()` klasy `String`, która w tym przypadku jest wygodniejsza od analogicznej metody klasy `Matcher`. Należy zauważyć, że obie operacje są wykonywane tylko raz, a zatem rezygnacja z wykorzystania skompilowanego wyrażenia reprezentowanego przez obiekt `Pattern` nie wiąże się z żadnymi negatywnymi konsekwencjami.

Metoda replaceFirst() zastępuje jedynie pierwszy odnaleziony fragment pasujący do wyrażania regularnego. Co więcej, ciągi zastępujące używane w wywołaniach metod replaceFirst() oraz replaceAll() są jedynie literałami, co sprawia, że metody te nie są przydatne w sytuacjach, gdy ciąg zastępujący ma być modyfikowany w trakcie operacji zastępowania. W takich przypadkach należy zastosować metodę appendReplacement(), która pozwala na wykonywanie dowolnego kodu podczas zastępowania. W powyższym przykładzie, posługując się metodą group(), pobieramy fragment ciągu wejściowego pasujący do grupy wyrażania regularnego, a następnie przetwarzamy go w trakcie tworzenia bufora sbuf. Przetwarzanie to polega na zapisaniu samogłosek wielkimi literami. Zazwyczaj działanie programu będzie polegać na wykonaniu wszystkich operacji zastąpienia i wywołaniu metody appendTail(); jeśli jednak chcemy zasymulować działanie metody replaceFirst() (lub zastąpić n pierwszych wystąpień wyrażenia), powinniśmy wykonać jedną operację zastąpienia i wywołać metodę appendTail(), aby pobrać pozostałą część ciągu wejściowego.

Metoda appendReplacement() pozwala także na bezpośrednie odwoływanie się w ciągu zastępującym do wybranych grup wyrażenia regularnego. Do tego celu wykorzystywane są sekwencje $g, gdzie g jest cyfrą określającą numer grupy. Z rozwiązania tego można korzystać w nieco prostszych przypadkach przetwarzania tekstów, a w powyższym przykładzie nie dałoby ono pożądanych rezultatów.

Metoda reset()

Istniejący obiekt Matcher można wykorzystać do przetworzenia nowej sekwencji znaków. W tym celu należy się posłużyć metodą reset():

```
//: strings/Resetting.java
import java.util.regex.*;

public class Resetting {
  public static void main(String[] args) throws Exception {
    Matcher m = Pattern.compile("[wftzb][aiey][snlr]")
      .matcher("filmu bali się wszyscy");
    while(m.find())
      System.out.print(m.group() + " ");
    System.out.println();
    m.reset("winni stali w szeregu");
    while(m.find())
      System.out.print(m.group() + " ");
  }
} /* Output:
fil bal zys
win tal zer
*///:~
```

Wywołanie metody reset() bez przekazania jakichkolwiek argumentów powoduje ustawienie obiektu Matcher na początku bieżącej sekwencji znaków.

Wyrażenia regularne i operacje wejścia-wyjścia Javy

W większości przedstawionych przykładów wyrażenia regularne służyły do obsługi statycznych ciągów znaków. Poniższy przykład pokazuje jeden ze sposobów wykorzystania wyrażeń regularnych do odnajdywania fragmentów zawartości pliku. Program *JGrep.java* jest wzorowany na uniksowym narzędziu *grep* i wymaga podania dwóch argumentów: nazwy pliku oraz wyrażenia regularnego. Program wyświetla każdy wiersz wskazanego pliku, w którym został odnaleziony fragment pasujący do podanego wyrażenia, oraz położenie tego fragmentu w wierszu.

```
// : strings/JGrep.java
// Bardzo prosta wersja programu "grep".
// {Args: JGrep.java "\\b[Ssct]\\w+"}
import java.util.regex.*;
import net.mindview.util.*;

public class JGrep {
  public static void main(String[] args) throws Exception {
    if(args.length < 2) {
      System.out.println("Stosowanie: java JGrep plik wyrażenieRegularne");
      System.exit(0);
    }
    Pattern p = Pattern.compile(args[1]);
    // Przegląd kolejnych wierszy pliku wejściowego:
    int index = 0;
    Matcher m = p.matcher("");
    for(String line : new TextFile(args[0])) {
      m.reset(line);
      while(m.find())
        System.out.println(index++ + ": " +
          m.group() + ": " + m.start());
    }
  }
} /* Output: (Sample)
0: strings: 4
1: Ssct: 26
2: class: 7
3: static: 9
4: String: 26
5: throws: 41
6: System: 6
7: Stosowanie: 26
8: System: 6
9: compile: 24
10: String: 8
11: System: 8
12: start: 31
13: Sample: 14
14: strings: 3
15: simple: 3
16: the: 3
17: Ssct: 3
18: class: 3
19: static: 3
20: String: 3
21: throws: 3
22: System: 3
```

```
23: System: 3
24: compile: 4
25: through: 4
26: the: 4
27: the: 4
28: String: 4
29: System: 4
30: start: 4
*///:~
```

Plik jest otwierany jako obiekt `TextFile` (klasę przedstawiam w rozdziale „Wejście-wyjście"), który umieszcza wiersze pliku w kontenerze `ArrayList`, którego z kolei można użyć w pętli foreach do przeglądania wierszy obiektu `TextFile`.

Dla każdego wiersza tekstu w pętli można by utworzyć osobny obiekt `Matcher`, ale lepiej na początku (jeszcze poza pętlą) utworzyć obiekt pusty i użyć metody `reset()` do kojarzenia kolejnych wierszy wejścia z obiektem `Matcher`. Wyniki są sprawdzane za pomocą metody `find()`.

Testowe argumenty wywołania programu powodują pobranie zawartości pliku *JGrep.java*, w której zostaną wyszukane słowa rozpoczynające się od `[Ssct]`.

O wyrażeniach regularnych możesz dowiedzieć się znacznie więcej z książki *Mastering Regular Expressions, 2nd Edition* autorstwa Jeffrey'a E. F. Friedla (O'Reilly, 2002)[7]. Znaczną liczbę artykułów wprowadzających do tego tematu można znaleźć w internecie, sporo można się też dowiedzieć z dokumentacji towarzyszącej językom takim jak Perl czy Python.

Ćwiczenie 15. Zmodyfikuj program *JGrep.java* w taki sposób, aby w wierszu wywołania można było podawać flagi, na przykład: `Pattern.CASE_INSENSITIVE` bądź `Pattern.MULTILINE` (5).

Ćwiczenie 16. Zmodyfikuj program *JGrep.java* w taki sposób, aby w wierszu wywołania można było podawać zarówno nazwy katalogów, jak i plików (jeśli zostanie podana nazwa katalogu, należy przeszukiwać wszystkie umieszczone w nim pliki). Podpowiedź: listę nazw plików można wygenerować przy użyciu poniższej instrukcji (5):

```
String[] nazwyPlików = new File(".").list();
```

Ćwiczenie 17. Napisz program, który wczyta plik kodu źródłowego w języku Java (nazwę pliku będziesz podawać w wierszu wywołania programu) i wypisze z niego wszystkie komentarze (8).

Ćwiczenie 18. Napisz program, który wczyta plik kodu źródłowego w języku Java (nazwę pliku będziesz podawać w wierszu wywołania programu) i wypisze z niego wszystkie literały ciągów znaków (8).

Ćwiczenie 19. Na podstawie dwóch poprzednich ćwiczeń napisz program, który przeanalizuje plik kodu źródłowego w języku Java i wypisze nazwy wszystkich klas wykorzystywanych w danym programie (8).

---

[7] W przekładzie polskim dostępna jest pierwsza edycja tej książki: *Wyrażenia regularne*, Helion, 2001 — *przyp. tłum.*

Skanowanie wejścia

Do niedawna wczytywanie danych podawanych na wejście standardowe w formie czytelnej dla użytkownika było cokolwiek uciążliwe. Zwykle polegało to na wczytaniu wiersza tekstu, podziału na elementy leksykalne i wyłuskaniu z nich (przy użyciu najróżniejszych metod klas Integer, Double itd.) oczekiwanych wartości, jak tutaj:

```
//: strings/SimpleRead.java
import java.io.*;

public class SimpleRead {
  public static BufferedReader input = new BufferedReader(
    new StringReader("Sir Robin z Camelot\n22 1.61803"));
  public static void main(String[] args) {
    try {
      System.out.println("Jak się nazywasz?");
      String name = input.readLine();
      System.out.println(name);
      System.out.println(
        "Ile masz lat? Jaka jest Twoja ulubiona liczba zmiennoprzecinkowa?");
      System.out.println("(Wejście: <wiek> <liczba>)");
      String numbers = input.readLine();
      System.out.println(numbers);
      String[] numArray = numbers.split(" ");
      int age = Integer.parseInt(numArray[0]);
      double favorite = Double.parseDouble(numArray[1]);
      System.out.format("Witaj, %s.\n", name);
      System.out.format("Za 5 lat będziesz miał ich %d.\n",
        age + 5);
      System.out.format("A moja ulubiona liczba zmiennoprzecinkowa to %f.",
        favorite / 2);
    } catch(IOException e) {
      System.err.println("Wyjątek wejścia-wyjścia");
    }
  }
} /* Output:
Jak się nazywasz?
Sir Robin z Camelot
Ile masz lat? Jaka jest Twoja ulubiona liczba zmiennoprzecinkowa?
(Wejście: <wiek> <liczba>)
22 1.61803
Witaj, Sir Robin z Camelot.
Za 5 lat będziesz miał ich 27.
A moja ulubiona liczba zmiennoprzecinkowa to 0,809015.
*///:~
```

Pole input wykorzystuje klasy z biblioteki java.io, które zostaną oficjalnie zaprezentowane dopiero w rozdziale „Wejście-wyjście". Klasa StringReader zamienia ciąg String na strumień dający się odczytywać, a strumień ten z kolei stanowi podstawę do utworzenia obiektu klasy BufferedReader — klasa ta posiada bowiem potrzebną nam metodę readLine(). W efekcie zawartość obiektu input może być wczytywana wierszami, tak jak to się odbywa w przypadku konsoli.

Metoda `readLine()` zwraca ciąg `String` zawierający kolejny wiersz wejścia. Takie podejście sprawdza się, kiedy porcje danych pokrywają się z wierszami. Kiedy jednak w jednym wierszu pojawią się dwie wartości wejściowe, sprawa się komplikuje — wiersz trzeba podzielić na dwa przetwarzane jako osobne wiersze wejścia. Podział odbywa się tu przy tworzeniu tablicy `numArray`, zauważ jednak, że metoda `split()` pojawiła się dopiero w J2SE1.4 — wcześniej trzeba było kombinować inaczej.

W Javie SE5 programista może złożyć znaczną część mitręgi przetwarzania wejścia na klasę `Scanner`:

```
//: strings/BetterRead.java
import java.util.*;

public class BetterRead {
  public static void main(String[] args) {
    Scanner stdin = new Scanner(SimpleRead.input);
    stdin.useLocale(Locale.ENGLISH);
    System.out.println("Jak się nazywasz?");
    String name = stdin.nextLine();
    System.out.println(name);
    System.out.println(
      "Ile masz lat? Jaka jest Twoja ulubiona liczba zmiennoprzecinkowa?");
    System.out.println("(wejście: <wiek> <liczba>)");
    int age = stdin.nextInt();
    double favorite = stdin.nextDouble();
    System.out.println(age);
    System.out.println(favorite);
    System.out.format("Witaj, %s.\n", name);
    System.out.format("Za 5 lat będziesz ich miał %d.\n",
      age + 5);
    System.out.format("A moja ulubiona liczba zmiennoprzecinkowa to %f.",
      favorite / 2);
  }
} /* Output:
Jak się nazywasz?
Sir Robin z Camelot
Ile masz lat? Jaka jest Twoja ulubiona liczba zmiennoprzecinkowa?
(wejście: <wiek> <liczba>)
22
1.61803
Witaj, Sir Robin z Camelot.
Za 5 lat będziesz ich miał 27.
A moja ulubiona liczba zmiennoprzecinkowa to 0,809015.
*///:~
```

Konstruktor klasy `Scanner` może przyjąć w wywołaniu obiekt wejściowy dowolnego rodzaju, w tym obiekt klasy `File` (o którym dowiemy się więcej w rozdziale „Wejście-wyjście"), `InputStream`, `String` albo (jak w przykładzie) dowolną implementację `Readable`, czyli interfejsu wprowadzonego w wydaniu SE5 jako opisującego typy „posiadające metodę `read()`". Do tej ostatniej kategorii zalicza się wykorzystana w przykładzie klasa `BufferedReader`.

Dzięki klasie `Scanner` wczytywanie, podział na leksemy i przetwarzanie są hermetyzowane w rozmaitych metodach `next...`. Zwykłe wywołanie `next()` zwraca następny leksem w postaci ciągu `String`; są też jednak wersje `next...` dla wszystkich typów elementarnych

(z wyjątkiem char), a także dla wartości reprezentowanych w Javie obiektami BigDecimal i BigInteger. Wszystkie metody next... są metodami blokującymi, co oznacza, że zwracają sterowanie do wywołującego dopiero wtedy, kiedy na wejściu pojawi się kompletny leksem. Każda z nich posiada też odpowiednią metodę hasNext..., która podgląda wejście i zwraca true w razie dostępności na wejściu następnego leksemu, właściwego dla danego typu danych.

Ciekawą różnicą pomiędzy dwoma poprzednimi przykładami jest brak bloku try dla wyjątku IOException w wersji *BetterRead.java*. Otóż klasa Scanner opiera swoje działanie między innymi na założeniu, że wyjątek IOException sygnalizuje fakt wyczerpania wejścia, więc te wyjątki są „połykane" przez klasę Scanner. W każdej chwili (jeśli to potrzebne) można jednak podejrzeć ostatnio przechwycony w niej wyjątek za pomocą metody ioException().

Ćwiczenie 20. Napisz klasę zawierającą pola int, long, float i double oraz pole String. Wyposaż ją w konstruktor, który będzie przyjmował pojedynczy argument typu String i przeszukiwał tak otrzymany ciąg w poszukiwaniu wartości dla poszczególnych pól. Dodaj do klasy metodę toString() i wykaż, że klasa działa poprawnie (2).

Separatory wartości wejściowych

Domyślnie klasa Scanner dzieli ciąg wejściowy w miejscach wystąpień znaków odstępu, ale można podać własny separator wartości, mający postać wyrażenia regularnego:

```
//: strings/ScannerDelimiter.java
import java.util.*;

public class ScannerDelimiter {
  public static void main(String[] args) {
    Scanner scanner = new Scanner("12, 42, 78, 99, 42");
    scanner.useDelimiter("\\s*,\\s*");
    while(scanner.hasNextInt())
      System.out.println(scanner.nextInt());
  }
} /* Output:
12
42
78
99
42
*///:~
```

W tym przykładzie rolę separatorów wartości wejściowych przy przetwarzaniu ciągu String odgrywały przecinki (otoczone dowolną liczbą znaków odstępów). Tę samą technikę można wykorzystać do wczytywania plików wartości oddzielanych przecinkami. Metodę useDelimiter(), ustawiającą nowe wyrażenie regularne separatora, uzupełnia metoda delimiter(), zwracająca obiekt Pattern bieżącego wyrażenia separującego wartości.

Skanowanie wejścia przy użyciu wyrażeń regularnych

Wejście można skanować nie tylko w poszukiwaniu wartości predefiniowanych typów podstawowych, ale i wartości formatowanych wedle własnych wzorców, co jest szczególnie pomocne przy wczytywaniu i przetwarzaniu bardziej złożonych danych. Poniższy przykład przetwarza rejestr ataków; podobny rejestr mógłby zostać wygenerowany przez zaporę sieciową:

```
//: strings/ThreatAnalyzer.java
import java.util.regex.*;
import java.util.*;

public class ThreatAnalyzer {
  static String threatData =
    "58.27.82.161@02/10/2005\n" +
    "204.45.234.40@02/11/2005\n" +
    "58.27.82.161@02/11/2005\n" +
    "58.27.82.161@02/12/2005\n" +
    "58.27.82.161@02/12/2005\n" +
    "[Następna sekcja rejestru z innym formatem danych]";
  public static void main(String[] args) {
    Scanner scanner = new Scanner(threatData);
    String pattern = "(\\d+[.]\\d+[.]\\d+[.]\\d+)@" +
      "(\\d{2}/\\d{2}/\\d{4})";
    while(scanner.hasNext(pattern)) {
      scanner.next(pattern);
      MatchResult match = scanner.match();
      String ip = match.group(1);
      String date = match.group(2);
      System.out.format("Atak dnia %s z %s\n", date,ip);
    }
  }
} /* Output:
Atak dnia 02/10/2005 z 58.27.82.161
Atak dnia 02/11/2005 z 204.45.234.40
Atak dnia 02/11/2005 z 58.27.82.161
Atak dnia 02/12/2005 z 58.27.82.161
Atak dnia 02/12/2005 z 58.27.82.161
*///:~
```

Jeśli korzystasz z metody `next()` i własnego wzorca, wzorzec ten jest dopasowywany do następnego elementu leksykalnego na wejściu. Wynik dopasowania jest udostępniany za pośrednictwem metody `match()`, jak widać powyżej — całość działa jak znane już nam dopasowywanie wyrażeń regularnych.

Skanowanie z użyciem wyrażenia regularnego wiąże się z jednym niebezpieczeństwem. Otóż wzorzec jest dopasowywany wyłącznie do następnego elementu leksykalnego, więc jeśli wyrażenie regularne wzorca zawiera separator, nie zostanie nigdy pomyślnie dopasowane.

Klasa StringTokenizer

Przed udostępnieniem wyrażeń regularnych (w J2SE1.4), a potem klasy Scanner (w Javie SE5), jedynym sposobem podziału ciągu na części był jego „rozbiór" za pomocą klasy `StringTokenizer`. Aktualnie jednak znacznie łatwiej i szybciej można osiągnąć te same efekty, stosując wyrażania regularne bądź klasę Scanner. Oto proste porównanie obu tych technik z zastosowaniem klasy `StringTokenizer`:

```
//: strings/ReplacingStringTokenizer.java
import java.util.*;

public class ReplacingStringTokenizer {
  public static void main(String[] args) {
    String input = "Ale ja jeszcze żyję! I dobrze się czuję!";
    StringTokenizer stoke = new StringTokenizer(input);
    while(stoke.hasMoreElements())
      System.out.print(stoke.nextToken() + " ");
    System.out.println();
    System.out.println(Arrays.toString(input.split(" ")));
    Scanner scanner = new Scanner(input);
    while(scanner.hasNext())
      System.out.print(scanner.next() + " ");
  }
} /* Output:
Ale ja jeszcze żyję! I dobrze się czuję!
[Ale, ja, jeszcze, żyję!, I, dobrze, się, czuję!]
Ale ja jeszcze żyję! I dobrze się czuję!
*///:~
```

Wyrażenia regularne i klasa Scanner pozwalają także na dzielenie ciągów znaków na części przy wykorzystaniu bardziej złożonych wzorców; uzyskanie podobnych możliwości przy bazowaniu na obiektach `StringTokenizer` jest znacznie trudniejsze.

Podsumowanie

W przeszłości narzędzia manipulowania ciągami znaków w języku Java miały postać szczątkową, ale w ostatnich wersjach języka można już cieszyć się dalece bardziej wyrafinowanymi mechanizmami, znanymi z innych języków programowania. Obsługę ciągów znaków można więc uznać za kompletną, choć nie zawsze doskonałą, choćby z uwagi na potencjalną nieefektywność konkatenacji i konieczność ręcznego stosowania klasy `StringBuilder`.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 14.

Informacje o typach

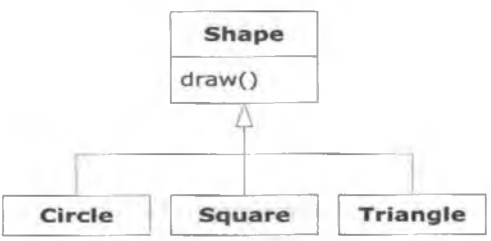
Informacje o typie w czasie wykonania (ang. run-time type information, RTTI) pozwalają na identyfikowanie typów i wykorzystywanie informacji o nich w czasie działania programu.

Mechanizm taki pozwala programiście korzystać z wiedzy na temat typów danych nie tylko w czasie kompilacji i stanowi bardzo silne narzędzie programistyczne. Jednak *potrzeba stosowania* mechanizmu RTTI odkrywa mnóstwo interesujących (i często zagmatwanych) kwestii projektowania obiektowo zorientowanego i rodzi fundamentalne pytanie: jak powinno się konstruować programy?

W tym rozdziale poznamy sposoby zdobywania informacji o obiektach i klasach w czasie działania programu. Istnieją dwa rodzaje tych sposobów: „tradycyjne" RTTI, które zakłada, że w czasie kompilacji i podczas działania dostępne są wszystkie typy oraz mechanizm „refleksji[1]" (ang. *reflection*) pozwalający na wykrycie informacji o klasie jedynie w czasie wykonania.

Potrzeba mechanizmu RTTI

Rozważmy znany już przykład hierarchii klas z wykorzystaniem polimorfizmu. Typem ogólnym jest klasa bazowa Shape, a specjalizowanymi typami pochodnymi są: `Circle`, `Square` oraz `Triangle`:



[1] Powszechnie przyjęte tłumaczenie dosłowne pojęcia *reflection* pokazuje jak wielki wpływ ma język angielski na powszechnie stosowany język informatyczny — *przyp. red.*

Jest to zwyczajny diagram hierarchii klas z klasą bazową na szczycie oraz klasami dziedziczącymi rozrastającymi się w dół. Podstawowym celem programowania zorientowanego obiektowo jest w większości przypadków manipulowanie referencjami do klasy bazowej (w tym przypadku Shape), więc jeżeli zdecydujemy się rozszerzyć program poprzez dodanie nowej klasy (na przykład Rhomboid dziedziczący z Shape), większość kodu pozostanie nienaruszona. W naszym przykładzie metodą wiązaną dynamicznie z klasy Shape jest metoda draw(), a zatem intencją programisty-klienta winno być wywołanie draw() poprzez referencję do ogólnego interfejsu Shape. Metoda ta jest przesłaniana we wszystkich klasach pochodnych i, ponieważ jest to metoda wiązana w sposób dynamiczny, zostanie wywołana właściwa metoda z klasy pochodnej, mimo korzystania z referencji do klasy Shape. To jest właśnie polimorfizm.

Tak więc zazwyczaj tworzy się konkretny obiekt (Circle, Square lub Triangle), rzutuje w górę na Shape (zapominając o konkretnym typie obiektu) i wykorzystuje referencję do anonimowego obiektu Shape w pozostałej części programu.

Hierarchię Shape można by oprogramować następująco:

```
//: typeinfo/Shapes.java
import java.util.*;

abstract class Shape {
  void draw() { System.out.println(this + ".draw()"); }
  abstract public String toString();
}

class Circle extends Shape {
  public String toString() { return "Circle"; }
}

class Square extends Shape {
  public String toString() { return "Square"; }
}

class Triangle extends Shape {
  public String toString() { return "Triangle"; }
}

public class Shapes {
  public static void main(String[] args) {
    List<Shape> shapeList = Arrays.asList(
      new Circle(), new Square(), new Triangle()
    );
    for(Shape shape : shapeList)
      shape.draw();
  }
} /* Output:
Circle.draw()
Square.draw()
Triangle.draw()
*///:~
```

Klasa bazowa zawiera metodę draw(), która pośrednio stosuje wywołanie toString() w celu wypisania identyfikatora klasy poprzez przekazanie this do System.out.println() (zauważ, że metoda toString() jest zadeklarowana jako abstrakcyjna, co wymusza jej

przesłonięcie w klasach pochodnych i równocześnie zapobiega tworzeniu egzemplarzy Shape). Jeśli w wyrażeniu konkatenacji ciągów znaków pojawi się obiekt, następuje automatyczne wywołanie metody toString() tego obiektu w celu uzyskania ciągu reprezentującego obiekt. Każda z klas pochodnych przesłania metodę toString() (z klasy Object) tak, że metoda draw() (dzięki wykorzystaniu polimorfizmu) wypisuje coś innego.

W powyższym przykładzie rzutowanie w górę następuje wtedy, gdy „kształt" jest umieszczany w kontenerze List<Shape>. Jednak podczas rzutowania w górę wstawiany obiekt traci wszystkie informacje *specyficzne dla jego konkretnego typu*. Dla kontenera są to tylko obiekty Shape.

Kiedy sięga się do elementu, kontener — który wewnętrznie przechowuje wszystko jako egzemplarze klasy Object — automatycznie dokonuje rzutowania z powrotem na typ Shape. Jest to najbardziej podstawowa z form RTTI, gdyż w Javie wszystkie rzutowania są sprawdzane pod względem poprawności w czasie wykonania. To właśnie dokładnie oznacza RTTI — w czasie wykonania rozpoznawany jest typ obiektu.

W tym przypadku rzutowanie RTTI jest tylko częściowe — Object jest rzutowany na Shape, a nie do Circle, Square czy Triangle. Jest tak, ponieważ jedyną rzeczą, którą *wiemy* w tym momencie, jest to, że kontener List<Shape> jest wypełniony obiektami klasy Shape. W czasie kompilacji jest to wymuszane tylko przez kontener i mechanizm typów ogólnych, ale podczas wykonania zapewnia nam to rzutowanie.

Teraz polimorfizm przejmuje obowiązki i konkretna metoda wywoływana wobec Shape jest ustalana w zależności od tego, czy referencja faktycznie odnosi się do obiektu Circle, Square czy Triangle. Tak właśnie powinno być — chcemy, by większość kodu wiedziała tylko tyle, ile trzeba, o *konkretnych* typach obiektów i korzystała z ogólnej reprezentacji rodziny obiektów (w tym przypadku Shape). W rezultacie kod będzie łatwiejszy do zapisania, odczytania i utrzymania, a projekt prostszy koncepcyjnie, w implementacji i zmianach. Zatem polimorfizm jest podstawową techniką programowania obiektowego.

Ale co w przypadku, kiedy mamy specjalne zadanie — łatwiejsze do rozwiązania, jeśli znamy dokładny typ referencji? Na przykład, przypuśćmy, że chcemy pozwolić użytkownikom na oznaczenie wszystkich figur dowolnego wybranego typu poprzez zmianę ich koloru na fioletowy. Dzięki temu mogą oni odnaleźć wszystkie trójkąty na ekranie poprzez ich podświetlenie. Albo załóżmy, że metoda musi dokonać obrotu wszystkich figur z listy. Obracanie okręgów nie ma jednak sensu, a zatem podczas wykonywania metody będziemy chcieli pominąć te obiekty. To właśnie realizuje RTTI — można spytać odwołanie do Shape o dokładny typ, na który wskazuje.

Obiekt Class

Aby zrozumieć, jak działa RTTI w Javie, trzeba najpierw poznać reprezentację informacji o typie w czasie wykonania. Służy do tego specjalny rodzaj obiektu, zwany *obiektem Class*, który zawiera informacje o klasie (czasami nazywa się go *metaklasą*). Tak naprawdę obiekt Class jest wykorzystywany do tworzenia wszystkich „normalnych" obiektów naszych klas. Java realizuje więc RTTI za pomocą obiektu Class. Klasa ta udostępnia też szereg innych sposobów użycia RTTI.

Dla każdej z klas, będących fragmentem naszego programu, istnieje obiekt typu `Class`. Jego stworzenie ma miejsce każdorazowo po napisaniu i kompilacji nowej klasy — tworzony jest także pojedynczy obiekt `Class` (który jest przechowywany w pliku o nazwie takiej jak klasa z rozszerzeniem *.class*). Aby utworzyć obiekt tej klasy, maszyna wirtualna Javy (ang. JVM — *Java Virtual Machine*) wykonująca program korzysta z podsystemu o nazwie *class loader*.

Ów podsystem może się w istocie składać z całego łańcucha „modułów ładujących", ale zawsze jest tylko jeden *pierwotny class loader*, wchodzący w skład implementacji maszyny wirtualnej. Ów pierwotny moduł ładujący klasy ładuje tak zwane *klasy zaufane*, w tym klasy interfejsu API języka Java — zazwyczaj przechowywane na dysku lokalnym. Zwykle obecność dodatkowych class loaderów jest zbędna, ale dla potrzeb specjalnych (na przykład ładowania klas w specjalny sposób, uwzględniający specyfikę obsługi aplikacji serwera WWW, czy ładowania klas z pobieraniem ich z sieci) można do łańcucha podpiąć dodatkowe moduły ładujące.

Wszystkie klasy są ładowane do maszyny wirtualnej dynamicznie, każda w momencie pierwszego jej użycia. W programie odpowiada to miejscu występowania pierwszego odwołania do statycznej składowej klasy. Okazuje się, że konstruktor jest również metodą statyczną klasy, mimo braku jawnego słowa kluczowego `static` przy definicji konstruktora. Dlatego tworzenie nowego obiektu klasy za pośrednictwem operatora `new` również liczy się jako odwołanie do statycznej składowej klasy.

Wynika z tego, że program w języku Java nie jest ładowany w całości jeszcze przed jego uruchomieniem — poszczególne jego części są doładowywane w czasie działania programu. To różni Javę od wielu innych języków programowania. Dynamiczne ładowanie klas owocuje zachowaniem, które w statycznie ładowanych językach, takich jak C++, jest trudne albo nawet niemożliwe do uzyskania.

Class loader w pierwszej kolejności sprawdza, czy obiekt `Class` dla danego typu jest już załadowany. Jeżeli nie, odszukuje plik *.class* według nazwy klasy (class loader inny niż pierwotny mógłby w tym momencie na przykład przeszukać nie system plików, ale bazę danych). W czasie ładowania kodu bajtowego klasy poszczególne bajty są weryfikowane w celu sprawdzenia, czy plik nie uległ uszkodzeniu i czy nie zawiera szkodliwego kodu (to jedna z linii obronnych mechanizmów bezpieczeństwa Javy).

Gdy obiekt `Class` znajdzie się już w pamięci, jest wykorzystywany do tworzenia wszystkich obiektów typu, który reprezentuje. Oto program demonstracyjny, który to udowodni:

```
//: typeinfo/SweetShop.java
// Analiza sposobu działania class loadera.
import static net.mindview.util.Print.*;

class Candy {
  static { print("Ładowanie klasy Candy"); }
}

class Gum {
  static { print("Ładowanie klasy Gum"); }
}

class Cookie {
  static { print("Ładowanie klasy Cookie"); }
```

```
}

public class SweetShop {
  public static void main(String[] args) {
    print("W metodzie main");
    new Candy();
    print("Po konstrukcji obiektu Candy");
    try {
      Class.forName("Gum");
    } catch(ClassNotFoundException e) {
      print("Nie można znaleźć klasy Gum");
    }
    print("Po wywołaniu Class.forName(\"Gum\")");
    new Cookie();
    print("Po konstrukcji obiektu Cookie");
  }
} /* Output:
W metodzie main
Ładowanie klasy Candy
Po konstrukcji obiektu Candy
Ładowanie klasy Gum
Po wywołaniu Class.forName("Gum")
Ładowanie klasy Cookie
Po konstrukcji obiektu Cookie
*///:~
```

Każda z klas `Candy`, `Gum` i `Cookie` posiada klauzulę `static`, która jest wykonywana podczas ładowania klasy po raz pierwszy. Każda z klas wypisuje informację, aby pokazać, kiedy dokładnie ma miejsce ładowanie takiej klasy. W metodzie głównej `main()` przy tworzeniu obiektów wypisywana jest informacja pomagająca wykryć moment ładowania.

Analizując wyniki, można się przekonać, że każdy obiekt `Class` jest ładowany dopiero w chwili gdy jest potrzebny, a instrukcje umieszczone w klauzuli `static` są wykonywane w momencie ładowania klasy.

Szczególnie interesujący jest wiersz:

```
Class.forName("Gum");
```

Wszystkie obiekty klas należą do klasy `Class`. Obiekt `Class` jest zwykłym obiektem Javy, dlatego można uzyskać i manipulować referencją do niego (to właśnie robi class loader). Jednym ze sposobów pozyskania referencji do obiektu `Class` jest statyczna metoda `forName()`, która pobiera obiekt `String` zawierający nazwę (przyjrzyj się pisowni i wielkim literom!) konkretnej klasy, do której chcemy pobrać referencję. Metoda zwraca referencję do obiektu `Class`, którą w tym przypadku ignorujemy. Metoda `forName()` wywoływana jest ze względu na jej efekt uboczny, którym jest załadowanie klasy `Gum`, oczywiście, jeśli program nie załadował jej wcześniej. Podczas procesu ładowania wykonywane są instrukcje umieszczone w klauzuli `static`.

W powyższym przykładzie, jeśli wywołanie metody `Class.forName()` zakończy się niepowodzeniem ze względu na brak ładowanej klasy, zostanie zgłoszony wyjątek `ClassNotFoundException`. W tym przypadku zgłoszenie wyjątku powoduje jedynie wyświetlenie informacji o błędzie i dalszą realizację programu, jednak w bardziej wyrafinowanych aplikacjach, wewnątrz procedury obsługi błędu, można podjąć próbę rozwiązania zaistniałej sytuacji.

Za każdym razem, kiedy chcesz się odwołać do informacji o typie w czasie wykonania, musisz zaopatrzyć się w referencję odpowiedniego obiektu typu `Class`. Można do tego wykorzystać metodę `Class.forName()`, bo wtedy można uzyskać referencję obiektu `Class` bez potrzeby posiadania egzemplarza interesującej nas klasy. Ale jeśli dysponujesz już obiektem danej klasy i chcesz dowiedzieć się czegoś o jego typie w czasie wykonania, powinieneś pobrać referencję `Class` za pośrednictwem metody wchodzącej w skład interfejsu klasy `Object`: `getClass()`. Metoda ta zwraca referencję `Class` reprezentującą właściwy typ obiektu, na rzecz którego została wywołana. Sama klasa `Class` udostępnia wiele interesujących metod; niektóre z nich prezentowane są w następnym programie przykładowym:

```
//: typeinfo/toys/ToyTest.java
// Testowanie klasy Class.
package typeinfo.toys;
import static net.mindview.util.Print.*;

interface HasBatteries {}
interface Waterproof {}
interface Shoots {}

class Toy {
  // Oznaczenie jako komentarz poniższego konstruktora domyślnego
  // spowoduje zgłoszenie wyjątku NoSuchMethodError z (*1*)
  Toy() {}
  Toy(int i) {}
}

class FancyToy extends Toy
implements HasBatteries, Waterproof, Shoots {
  FancyToy() { super(1); }
}

public class ToyTest {
  static void printInfo(Class cc) {
    print("Nazwa klasy: " + cc.getName() +
      ", interfejs? [" + cc.isInterface() + "]");
    print("Nazwa prosta: " + cc.getSimpleName());
    print("Nazwa kanoniczna: " + cc.getCanonicalName());
  }
  public static void main(String[] args) {
    Class c = null;
    try {
      c = Class.forName("typeinfo.toys.FancyToy");
    } catch(ClassNotFoundException e) {
      print("Nie można znaleźć klasy FancyToy");
      System.exit(1);
    }
    printInfo(c);
    for(Class face : c.getInterfaces())
      printInfo(face);
    Class up = c.getSuperclass();
    Object obj = null;
    try {
      // Wymaga konstruktora domyślnego:
      obj = up.newInstance();
    } catch(InstantiationException e) {
```

```
        print("Nie można utworzyć egzemplarza");
        System.exit(1);
      } catch(IllegalAccessException e) {
        print("Brak dostępu");
        System.exit(1);
      }
      printInfo(obj.getClass());
    }
  } /* Output:
  Nazwa klasy: typeinfo.toys.FancyToy, interfejs? [false]
  Nazwa prosta: FancyToy
  Nazwa kanoniczna: typeinfo.toys.FancyToy
  Nazwa klasy: typeinfo.toys.HasBatteries, interfejs? [true]
  Nazwa prosta: HasBatteries
  Nazwa kanoniczna: typeinfo.toys.HasBatteries
  Nazwa klasy: typeinfo.toys.Waterproof, interfejs? [true]
  Nazwa prosta: Waterproof
  Nazwa kanoniczna: typeinfo.toys.Waterproof
  Nazwa klasy: typeinfo.toys.Shoots, interfejs? [true]
  Nazwa prosta: Shoots
  Nazwa kanoniczna: typeinfo.toys.Shoots
  Nazwa klasy: typeinfo.toys.Toy, interfejs? [false]
  Nazwa prosta: Toy
  Nazwa kanoniczna: typeinfo.toys.Toy
  *///:~
```

Klasa `FancyToy` dziedziczy po `Toy` i implementuje interfejsy `HasBatteries`, `Waterproof` i `Shoots`. W metodzie `main()` następuje utworzenie i zainicjalizowanie referencji `Class` dla klasy `FancyToy` — zwraca ją metoda `forName()` w odpowiednim bloku `try`. Zauważ, że w ciągu znaków przekazywanym do `forName()` trzeba użyć pełnej kwalifikowanej nazwy klasy (z nazwą pakietu włącznie).

Metoda `printInfo()` produkuje kwalifikowaną nazwę klasy za pomocą metody `getName()`, a do uzyskania nazwy (bez nazwy pakietu) i pełnej kwalifikowanej nazwy klas prostych wywołuje metody `getSimpleName()` i `getCanonicalName()` (wprowadzone w Javie SE5). Z kolei metoda `isInterface()`, zgodnie ze swoją nazwą, informuje o tym, czy dany obiekt `Class` reprezentuje interfejs. Jak widać, obiekt `Class` dla danej klasy zawiera komplet informacji o typie.

Metoda `Class.getInterfaces()`, wywoływana w `main()`, zwraca tablicę obiektów `Class` reprezentujących interfejsy implementowane przez klasę badanego obiektu.

Posiadając egzemplarz `Class`, można zapytać o klasę bazową (bezpośrednią) badanej klasy — służy do tego metoda `getSuperClass()`. Metoda ta zwraca referencję `Class`, którą można analogicznie zapytać o jej klasę bazową — w ten sposób można w czasie wykonania przejrzeć całą hierarchię klas.

Metoda `Class.newInstance()` to sposób implementacji „konstruktora wirtualnego", dzięki któremu można wyrazić chęć utworzenia obiektu mimo braku wiedzy o jego konkretnym typie. W ostatnim przykładzie up to referencja `Class`, która w czasie kompilacji nie niesie z sobą żadnych dodatkowych informacji o typie, a utworzenie nowego egzemplarza tworzy referencję `Object`. Ale ta referencja odnosi się do obiektu klasy `Toy`. Zanim będzie można wysyłać do niej jakiekolwiek komunikaty poza tymi akceptowanymi przez klasę `Object`, trzeba zbadać właściwy typ obiektu i dokonać stosownego rzutowania. Do

tego klasa obiektu tworzonego przez wywołanie `newInstance()` musi posiadać konstruktor domyślny. W dalszej części rozdziału zobaczysz, w jaki sposób tworzyć dynamicznie obiekty klas przy użyciu dowolnie wybranego konstruktora, a to za sprawą *refleksji*.

Ćwiczenie 1. W programie *ToyTest.java* umieść w komentarzu konstruktor domyślny klasy `Toy` i wyjaśnij, co się dzieje (1).

Ćwiczenie 2. Włącz nowy rodzaj interfejsu do *ToyTest.java* i zweryfikuj, czy jest poprawnie wykrywany i wypisywany (2).

Ćwiczenie 3. Dodaj klasę rombu o nazwie `Rhomboid` do przykładu *Shapes.java*. Stwórz obiekt `Rhomboid`, zrzutuj go w górę na `Shape`, a potem z powrotem w dół na `Rhomboid`. Spróbuj rzutować w dół na typ `Circle` i zaobserwuj, co się stanie (2).

Ćwiczenie 4. Zmodyfikuj ćwiczenie 3. tak, by wykorzystać `instanceof` do sprawdzania typu przed wykonaniem rzutowania w dół (2).

Ćwiczenie 5. Zaimplementuj metodę obracającą figurę `rotate(Shape)` tak, by sprawdzała, czy przypadkiem nie ma obrócić okręgu `Circle` (i, jeżeli tak jest, nie wykonywała tej operacji) (3).

Ćwiczenie 6. Zmień *Shapes.java*, aby można było „oznaczyć" (ustawić znacznik) wszystkie figury jakiegoś określonego typu. Metoda `toString()` dla każdej z figur pochodnych powinna wskazywać, czy dana figura jest „oznaczona" (4).

Ćwiczenie 7. Zmień *SweetShop.java*, aby każdy rodzaj tworzenia obiektu był kontrolowany przez argumenty z wiersza poleceń. To znaczy, jeżeli wierszem poleceń jest `java SweetShop Candy`, to tworzony jest tylko obiekt `Candy`. Zauważ, w jaki sposób można kontrolować, które obiekty `Class` są ładowane z wiersza poleceń (3).

Ćwiczenie 8. Napisz metodę, która pobiera obiekt i rekurencyjnie wypisuje wszystkie klasy z jego hierarchii (5).

Ćwiczenie 9. Zmodyfikuj ćwiczenie 8. tak, aby stosować również metodę `Class.getDeclaredFields()` do wyświetlania informacji o polach składowych klasy (5).

Ćwiczenie 10. Napisz program rozpoznający, czy tablica znaków jest typem podstawowym czy prawdziwym obiektem (3).

Literały Class

Java umożliwia uzyskanie odwołania do obiektu `Class` w inny sposób — poprzez zastosowanie *literału klasy*. W powyższym programie wyglądałoby to tak:

```
Gum.class
```

Jest nie tylko prostsze, ale również bezpieczniejsze, ponieważ sprawdzenie poprawności kodu następuje w czasie kompilacji (przez co nie trzeba umieszczać go w bloku `try`). Ponieważ eliminujemy wywołanie metody, jest to także bardziej wydajne.

Literały klasy działają ze zwykłymi klasami, jak również z interfejsami, tablicami i typami podstawowymi. Dodatkowo istnieje standardowe pole o nazwie TYPE, określone dla każdej z klas opakowujących typy podstawowe. Pole TYPE zwraca referencję do obiektu Class skojarzonego typu podstawowego w następujący sposób:

...jest równoważne z...	
boolean.class	Boolean.TYPE
char.class	Character.TYPE
byte.class	Byte.TYPE
short.class	Short.TYPE
int.class	Integer.TYPE
long.class	Long.TYPE
float.class	Float.TYPE
double.class	Double.TYPE
void.class	Void.TYPE

Osobiście preferuję stosowanie wersji .class, jeśli tylko jest to możliwe, ponieważ są one bardziej zgodne z normalnymi klasami.

Warto zaznaczyć, że tworzenie referencji obiektu Class za pośrednictwem literału .class nie oznacza automatycznego zainicjalizowania obiektu Class. Przygotowanie obiektu do użycia odbywa się w trzech krokach:

1. *Ładowanie* realizowane przez class loader. Polega na odszukaniu kodu bajtowego (zwykle — ale niekoniecznie — przechowywanego na dysku twardym w zasięgu zmiennej środowiskowej CLASSPATH) i utworzeniu na jego podstawie obiektu Class.
2. *Konsolidacja* polegająca na weryfikacji kodu bajtowego klasy, przydzieleniu pamięci dla pól statycznych i — w razie potrzeby — rozstrzygnięciu wszystkich odwołań do innych klas.
3. *Inicjalizacja*. Jeśli klasa ma klasę bazową, następuje inicjalizacja tej ostatniej. Potem dochodzi do wykonania inicjalizatorów składowych statycznych i statycznych bloków inicjalizacji.

Inicjalizacja jest opóźniana do czasu pojawienia się pierwszego odwołania do metody statycznej (konstruktor też jest niejawnie statyczny) albo odwołania do statycznego pola niebędącego stałą:

```
// : typeinfo/ClassInitialization.java
import java.util.*;

class Initable {
  static final int staticFinal = 47;
  static final int staticFinal2 =
    ClassInitialization.rand.nextInt(1000);
  static {
    System.out.println("Inicjalizacja klasy Initable");
  }
}
```

```
class Initable2 {
  static int staticNonFinal = 147;
  static {
    System.out.println("Inicjalizacja klasy Initable2");
  }
}

class Initable3 {
  static int staticNonFinal = 74;
  static {
    System.out.println("Inicjalizacja klasy Initable3");
  }
}

public class ClassInitialization {
  public static Random rand = new Random(47);
  public static void main(String[] args) throws Exception {
    Class initable = Initable.class;
    System.out.println("Po utworzeniu referencji Initable");
    // Nie powoduje inicjalizacji:
    System.out.println(Initable.staticFinal);
    // Powoduje inicjalizację:
    System.out.println(Initable.staticFinal2);
    // Powoduje inijcalizację:
    System.out.println(Initable2.staticNonFinal);
    Class initable3 = Class.forName("Initable3");
    System.out.println("Po utworzeniu referencji Initable3");
    System.out.println(Initable3.staticNonFinal);
  }
} /* Output:
Po utworzeniu referencji Initable
47
Inicjalizacja klasy Initable
258
Inicjalizacja klasy Initable2
147
Inicjalizacja klasy Initable3
Po utworzeniu referencji Initable3
74
*///:~
```

Zasadniczo inicjalizacja jest opóźniana tak długo, jak to możliwe. Utworzenie referencji `initable` ujawnia, że użycie zapisu `.class` do pozyskania referencji klasy nie oznacza bynajmniej inicjalizacji klasy, ale już wywołanie metody `Class.forName()` w celu wygenerowania referencji `Class` wymusza natychmiastową inicjalizację klasy — widać to przy okazji tworzenia `initable3`.

Jeśli wartość statyczna i finalna, jak `Initable.staticFinal`, jest „stałą czasu kompilacji", to wartość tą można odczytywać bez wymuszania inicjalizacji klasy `Initable`. Ale samo oznaczenie pola klasy jako statycznego i finalnego nie gwarantuje tego zachowania: odwołanie do `Initable.staticFinal2` wymusza inicjalizację klasy, bo pole to nie może być stałą czasu kompilacji.

Jeśli pole statyczne nie jest równocześnie polem finalnym, odwołanie do niego zawsze wymusza konsolidację (przydział pamięci dla pola) i inicjalizację jeszcze przed odczytem — efekt ten widać na przykładzie odwołania do `Initable2.staticNonFinal`.

Referencje klas uogólnionych

Referencja `Class` odnosi się do obiektu `Class`, który służy do tworzenia egzemplarzy danej klasy i zawiera całość kodu metod dla tych egzemplarzy. Obiekt ten zawiera też wszystkie składowe statyczne klasy. Referencja `Class` określa dokładny typ tego, do czego się odnosi: do obiektu klasy `Class`.

Projektanci Javy SE5 dostrzegli tu okazję do umożliwienia — za pomocą składni charakterystycznej dla typów ogólnych — ograniczenia typu obiektu `Class`, do którego odnosi się referencja `Class`. Obie składnie z poniższego przykładu są poprawne:

```
//: typeinfo/GenericClassReferences.java

public class GenericClassReferences {
  public static void main(String[] args) {
    Class intClass = int.class;
    Class<Integer> genericIntClass = int.class;
    genericIntClass = Integer.class; // To samo
    intClass = double.class;
    // genericIntClass = double.class; // Niedozwolone
  }
} ///:~
```

Zwykła referencja klasy nie spowoduje ostrzeżenia kompilatora. Ale widać, że ta zwykła referencja może zostać przestawiona na dowolny inny obiekt typu `Class`, podczas gdy referencja klasy z uogólnieniem może być skojarzona jedynie z obiektem typu zgodnego z deklaracją referencji. Składnia uogólnienia pozwala kompilatorowi na zastosowanie dodatkowej kontroli typów.

A co, gdybyśmy zechcieli nieco rozluźnić ograniczenia? Zdawałoby się, że można zapisać coś takiego:

```
Class<Number> genericNumberClass = int.class;
```

Wydaje się, że ma to sens, bo `Integer` dziedziczy po `Number`. Ale zapis taki nie zadziała, bowiem obiekt `Class` dla typu `Integer` nie jest bynajmniej podtypem obiektu `Class` dla typu `Number` (różnica jest dość subtelna — zajmiemy się nią ponownie w rozdziale „Typy ogólne").

Do rozluźniania ograniczeń przy używaniu referencji `Class` z typami ogólnymi stosowany jest *symbol wieloznaczny*, który jest częścią specyfikacji typów ogólnych Javy. Symbol wieloznaczny ma postać znaku zapytania (?) i oznacza „cokolwiek". Możemy więc w poprzednim przykładzie uzupełnić zwykłą referencję `Class` symbolem wieloznacznym i uzyskać identyczny efekt:

```
//: typeinfo/WildcardClassReferences.java

public class WildcardClassReferences {
  public static void main(String[] args) {
    Class<?> intClass = int.class;
    intClass = double.class;
  }
} ///:~
```

W Javie SE5 zapis `Class<?>` jest preferowany wobec zwykłego `Class`, mimo że — jak dowodzi powyższy przykład — są to zapisy równoważne. Zaletą stosowania uogólnienia z symbolem wieloznacznym `Class<?>` jest jawna sygnalizacja tego, że uogólnienie referencji na dowolne typy jest zamierzone, a nie jest np. skutkiem ignorancji. Po prostu świadomie wybieramy brak ograniczenia co do typu.

Aby utworzyć referencję `Class`, ograniczoną do wybranego typu *lub dowolnych jego podtypów*, należy *połączyć* symbol wieloznaczny ze słowem kluczowym extends. Zamiast zwykłego `Class<Number>` stosujemy więc coś takiego:

```
//: typeinfo/BoundedClassReferences.java

public class BoundedClassReferences {
  public static void main(String[] args) {
    Class<? extends Number> bounded = int.class;
    bounded = double.class;
    bounded = Number.class;
    // Albo cokolwiek, co dziedziczy po Number.
  }
} ///:~
```

Motywacją dla uzupełnienia referencji `Class` o składnię charakterystyczną dla typów ogólnych było jedynie umożliwienie kontroli typów podczas kompilacji, tak aby w razie czego można było nieco wcześniej wykryć błąd. Co prawda przy zwykłych referencjach `Class` trudno o błąd, ale jeśli zdarzy się pomyłka, dowiesz się o niej dopiero w czasie wykonania, co może być mało wygodne.

Oto przykład używający składni typów ogólnych. Zachowuje on referencję klasy, a potem generuje listę wypełnioną obiektami generowanymi za pomocą metody `newInstance()`.

```
//: typeinfo/FilledList.java
import java.util.*;

class CountedInteger {
  private static long counter;
  private final long id = counter++;
  public String toString() { return Long.toString(id); }
}

public class FilledList<T> {
  private Class<T> type;
  public FilledList(Class<T> type) { this.type = type; }
  public List<T> create(int nElements) {
    List<T> result = new ArrayList<T>();
    try {
      for(int i = 0; i < nElements; i++)
        result.add(type.newInstance());
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
    return result;
  }
  public static void main(String[] args) {
    FilledList<CountedInteger> fl =
      new FilledList<CountedInteger>(CountedInteger.class);
    System.out.println(fl.create(15));
  }
```

```
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
*///:~
```

Zauważ, że klasa musi zakładać, że wszelkie typy, na których operuje, posiadają konstruktor domyślny (bezargumentowy) — w przeciwnym razie dojdzie do zgłoszenia wyjątku. Kompilator nie wystosuje przy kompilacji powyższego programu żadnego ostrzeżenia.

Ciekawie robi się kiedy używamy składni typów ogólnych dla obiektów `Class`: metoda `newInstance()` zwróci tym razem obiekt dokładnego typu, a nie po prostu egzemplarz `Object`, jak w przykładzie *ToyTest.java*. To nieco ogranicza:

```
//: typeinfo/toys/GenericToyTest.java
// Testowanie klasy Class.
package typeinfo.toys;

public class GenericToyTest {
  public static void main(String[] args) throws Exception {
    Class<FancyToy> ftClass = FancyToy.class;
    // Generuje obiekt dokładnego typu:
    FancyToy fancyToy = ftClass.newInstance();
    Class<? super FancyToy> up = ftClass.getSuperclass();
    // Nie da się skompilować:
    // Class<Toy> up2 = ftClass.getSuperclass();
    // Zwraca jedynie egzemplarz Object:
    Object obj = up.newInstance();
  }
} ///:~
```

Kiedy dobieramy się do nadklasy, kompilator pozwala jedynie wyrazić, że referencja nadklasy ma odnosić się do „jakiejś klasy bazowej wobec `FancyToy`" — jak w wyrażeniu `Class<? super FancyToy>`. Nie przyjmie natomiast zapisu `Class<Toy>`. Nieco to dziwne, bo metoda `getSuperClass()` zwraca *klasę* bazową (nie interfejs) i kompilator zna tę klasę w czasie kompilacji — w tym przypadku jest to dokładnie `Toy.class`, a nie „jakaś" nadklasa `FancyToy`. Tak czy inaczej wartością zwracaną z `up.newInstance()` nie jest obiekt właściwego typu, a najzwyklejszy egzemplarz `Object`.

Nowa składnia rzutowania

Java SE5 proponuje też nową składnię rzutowania wykorzystywaną z referencjami `Class`, w postaci metody `cast()`:

```
//: typeinfo/ClassCasts.java

class Building {}
class House extends Building {}

public class ClassCasts {
  public static void main(String[] args) {
    Building b = new House();
    Class<House> houseType = House.class;
    House h = houseType.cast(b);
    h = (House)b; // ... albo po prostu tak.
  }
} ///:~
```

Metoda `cast()` przyjmuje w wywołaniu obiekt i rzutuje go na typ referencji `Class`. Oczywiście, jeśli spojrzeć na powyższy kod, to nowa składnia zdaje się wielce kłopotliwa, zwłaszcza w porównaniu z ostatnim wierszem metody `main()`, gdzie takie samo rzutowanie odbywa się klasycznie. Nowa składnia rzutowania przydaje się tam, gdzie po prostu *nie można* użyć zwykłego rzutowania. Zdarza się to przy pisaniu kodu uogólnionego (któremu poświęcony jest w całości rozdział „Typy ogólne"), kiedy do dyspozycji jest referencja `Class`, która ma posłużyć do rzutowania nań w czasie późniejszym. Ale to rzadkość — w całej bibliotece Javy SE5 znalazłem zaledwie jedno użycie metody `cast()` (w `com.sun.mirror.util.DeclarationFilter`).

Inną nowością SE5, której *w ogóle* nie używa biblioteka, jest metoda `Class.asSubclass()` pozwalająca na rzutowanie obiektu klasy na bardziej konkretny typ.

Sprawdzanie przed rzutowaniem

Jak dotąd poznałeś następujące postacie RTTI:

1. Klasyczne rzutowanie, np. `(Shape)`, które stosuje RTTI, aby upewnić się, że rzutowanie jest poprawne i zgłasza wyjątek `ClassCastException` w przypadku próby złego rzutowania.
2. Obiekt `Class` reprezentujący typ obiektu. Obiekt ten może być wypytywany o przydatne dla nas informacje w czasie wykonania programu.

W języku C++ klasyczne rzutowanie `(Shape)` *nie* używa RTTI. Po prostu mówi kompilatorowi, aby traktował obiekt jak nowy typ. W Javie, która przeprowadza kontrolę typu, rzutowanie to jest często nazywane „rzutowaniem w dół bezpiecznym dla typu". Przyczyną terminu „rzutowanie w dół" jest historyczna umowa dotycząca diagramu hierarchii klas. Skoro rzutowanie z klasy `Circle` na klasę `Shape` jest rzutowaniem w górę, to rzutowanie z `Shape` do `Circle` jest rzutowaniem w dół. Jednak wiemy, że okrąg także jest figurą, i kompilator swobodnie pozwala na przypisanie z rzutowaniem w górę, nie wymagając żadnej specjalnej składni. Ale kompilator *nie może* wiedzieć, czym dany obiekt `Shape` jest w rzeczywistości — może być właśnie obiektem typu `Shape`, ale równie dobrze `Circle`, `Suqare` czy `Triangle` albo jeszcze innym. W czasie kompilacji kompilator widzi jedynie typ `Shape`. Dlatego nie zezwoli na wykonanie przypisania z rzutowaniem w dół bez jawnego rzutowania, które ma dowodzić, że programista jest w posiadaniu dodatkowych informacji, dzięki którym wie, że rzutowany obiekt jest właśnie tego, a nie innego typu (kompilator *mimo to* sprawdzi, czy rzutowanie w dół jest zasadne, więc nie pozwoli na rzutowanie na coś innego, niż podtyp danego typu).

Istnieje trzecia postać RTTI w Javie. Jest to słowo kluczowe `instanceof`, które informuje, czy obiekt jest egzemplarzem podanego typu. Zwraca ono wartość logiczną, stąd stosuje się je w postaci pytania, jak np.:

```
if(x instanceof Dog)
  ((Dog)x).bark();
```

Warunek w instrukcji `if` sprawdza, czy obiekt `x` należy do klasy `Dog`, *zanim* nastąpi rzutowanie na `Dog`. Użycie `instanceof` przed rzutowaniem w dół jest istotne, jeśli nie posiadamy innych danych na temat typu obiektu — inaczej skończy się wyjątkiem `ClassCastException`.

Zazwyczaj będziemy szukać jednego typu obiektów (np. trójkątów, aby pokolorować je na fioletowo), ale można łatwo poznać *wszystkie* z obiektów, stosując `instanceof`. Przypuśćmy na przykład, że dysponujemy rodziną klas zwierząt domowych `Pet` (i ich właścicieli, którzy przydadzą się w następnym przykładzie). Otóż każdy osobnik (`Individual`) w hierarchii posiada własny identyfikator i, ewentualnie, imię. Choć prezentowane dalej klasy dziedziczą po klasie `Individual`, sama klasa jest cokolwiek złożona, a jej omówienie zostanie odłożone do rozdziału „Kontenery z bliska". Jak widać, kod klasy `Individual` jest nam w tym momencie całkowicie zbędny — wystarczy wiedzieć, że można tworzyć jej obiekty z imieniem albo bez niego i że każdy egzemplarz `Individual` posiada metodę `id()` zwracającą unikatowy identyfikator tegoż egzemplarza (generowany na bazie zliczania obiektów). Klasa ma też metodę `toString()`; jeśli egzemplarz wyposażymy w imię, metoda ta zwróci ciąg imienia; dla nienazwanych egzemplarzy `Individual` metoda `toString()` zwróci jedynie prostą nazwę typu.

Oto hierarchia klas dziedziczących po `Individual`:

```
//: typeinfo/pets/Person.java
package typeinfo.pets;

public class Person extends Individual {
  public Person(String name) { super(name); }
} ///:~

//: typeinfo/pets/Pet.java
package typeinfo.pets;

public class Pet extends Individual {
  public Pet(String name) { super(name); }
  public Pet() { super(); }
} ///:~

//: typeinfo/pets/Dog.java
package typeinfo.pets;

public class Dog extends Pet {
  public Dog(String name) { super(name); }
  public Dog() { super(); }
} ///:~

//: typeinfo/pets/Mutt.java
package typeinfo.pets;

public class Mutt extends Dog {
  public Mutt(String name) { super(name); }
  public Mutt() { super(); }
} ///:~

//: typeinfo/pets/Pug.java
package typeinfo.pets;

public class Pug extends Dog {
  public Pug(String name) { super(name); }
  public Pug() { super(); }
} ///:~
```

```
//: typeinfo/pets/Cat.java
package typeinfo.pets;

public class Cat extends Pet {
  public Cat(String name) { super(name); }
  public Cat() { super(); }
} ///:~

//: typeinfo/pets/EgyptianMau.java
package typeinfo.pets;

public class EgyptianMau extends Cat {
  public EgyptianMau(String name) { super(name); }
  public EgyptianMau() { super(); }
} ///:~

//: typeinfo/pets/Manx.java
package typeinfo.pets;

public class Manx extends Cat {
  public Manx(String name) { super(name); }
  public Manx() { super(); }
} ///:~

//: typeinfo/pets/Cymric.java
package typeinfo.pets;

public class Cymric extends Manx {
  public Cymric(String name) { super(name); }
  public Cymric() { super(); }
} ///:~

//: typeinfo/pets/Rodent.java
package typeinfo.pets;

public class Rodent extends Pet {
  public Rodent(String name) { super(name); }
  public Rodent() { super(); }
} ///:~

//: typeinfo/pets/Rat.java
package typeinfo.pets;

public class Rat extends Rodent {
  public Rat(String name) { super(name); }
  public Rat() { super(); }
} ///:~

//: typeinfo/pets/Mouse.java
package typeinfo.pets;

public class Mouse extends Rodent {
  public Mouse(String name) { super(name); }
  public Mouse() { super(); }
} ///:~

//: typeinfo/pets/Hamster.java
package typeinfo.pets;
```

```
public class Hamster extends Rodent {
  public Hamster(String name) { super(name); }
  public Hamster() { super(); }
} ///:~
```

Teraz potrzebujemy sposobu losowego tworzenia różnych typów zwierzaków i — dla wygody — tablic i list obiektów Pet. Aby tak zmontowane narzędzie poddawało się ewolucji na przestrzeni kilku przykładów, zdefiniujemy je w postaci abstrakcyjnej klasy bazowej:

```
//: typeinfo/pets/PetCreator.java
// Tworzy losowe sekwencje obiektów Pet.
package typeinfo.pets;
import java.util.*;

public abstract class PetCreator {
  private Random rand = new Random(47);
  // Lista (List) różnych typów obiektów Pet:
  public abstract List<Class<? extends Pet>> types();
  public Pet randomPet() { // Tworzenie losowego obiektu Pet
    int n = rand.nextInt(types().size());
    try {
      return types().get(n).newInstance();
    } catch(InstantiationException e) {
      throw new RuntimeException(e);
    } catch(IllegalAccessException e) {
      throw new RuntimeException(e);
    }
  }
  public Pet[] createArray(int size) {
    Pet[] result = new Pet[size];
    for(int i = 0; i < size; i++)
      result[i] = randomPet();
    return result;
  }
  public ArrayList<Pet> arrayList(int size) {
    ArrayList<Pet> result = new ArrayList<Pet>();
    Collections.addAll(result, createArray(size));
    return result;
  }
} ///:~
```

Abstrakcyjna metoda `types()` zdaje się w zadaniu pozyskania listy obiektów `Class` na klasę pochodną (to wariacja na temat wzorca projektowego *Template Method*). Zauważ, że typ klasy został określony jako „cokolwiek wyprowadzonego z klasy Pet", więc obiekty generowane przez `newInstance()` nie będą wymagać rzutowania. Metoda `randomPet()` odwołuje się do losowych indeksów listy i wykorzystuje tak wybrane klasy do tworzenia nowych egzemplarzy tychże klas właśnie za pomocą wywołania `Class.newInstance()`. Z usług metody `randomPet()` korzysta metoda `createArray()` wypełniająca tablicę; ta z kolei jest wykorzystywana w metodzie `arrayList()`.

Wywołanie `newInstance()` może spowodować zgłoszenie wyjątków dwojakiego typu — widać je w klauzulach `catch` umieszczonych za blokami `try`. Także w tym przypadku nazwy wyjątków w stosunkowo jasny sposób opisują przyczyny problemów (wyjątek `IllegalAccessException` jest związany z naruszeniem zasad bezpieczeństwa Javy; tu jest zgłaszany, kiedy konstruktor domyślny jest konstruktorem prywatnym).

Wyprowadzając podklasę PetCreator, trzeba jedynie udostępnić listę typów zwierzaków, które mają być uwzględniane w metodzie randomPet() i innych metodach. Metoda types() będzie zwyczajnie zwracać referencję listy statycznej. Oto implementacja bazująca na forName():

```
//: typeinfo/pets/ForNameCreator.java
package typeinfo.pets;
import java.util.*;

public class ForNameCreator extends PetCreator {
  private static List<Class<? extends Pet>> types =
    new ArrayList<Class<? extends Pet>>();
  // Typy, które mają być uwzględniane przy losowaniu:
  private static String[] typeNames = {
    "typeinfo.pets.Mutt",
    "typeinfo.pets.Pug",
    "typeinfo.pets.EgyptianMau",
    "typeinfo.pets.Manx",
    "typeinfo.pets.Cymric",
    "typeinfo.pets.Rat",
    "typeinfo.pets.Mouse",
    "typeinfo.pets.Hamster"
  };
  @SuppressWarnings("unchecked")
  private static void loader() {
    try {
      for(String name : typeNames)
        types.add(
          (Class<? extends Pet>)Class.forName(name));
    } catch(ClassNotFoundException e) {
      throw new RuntimeException(e);
    }
  }
  static { loader(); }
  public List<Class<? extends Pet>> types() {return types;}
} ///:~
```

Metoda loader() tworzy listę obiektów Class za pomocą metody Class.forName(). Może to spowodować wyjątek ClassNotFoundException, co nie powinno dziwić, skoro w roli argumentu przekazujemy ciąg String, którego zawartości nie da się zweryfikować w czasie kompilacji. Ponieważ klasy Pet znajdują się w pakiecie typeinfo, w odwołaniach do klas musi występować nazwa pakietu.

Do utworzenia listy obiektów Class konieczne jest rzutowanie powodujące ostrzeżenie kompilacji. Metoda loader() jest definiowana oddzielnie, a potem umieszczana w klauzuli inicjalizacji statycznej — a to dlatego, że w tej klauzuli nie można umieścić wprost adnotacji @SuppressWarnings.

Do zliczania obiektów Pet będzie nam potrzebne narzędzie rejestrujące ilość egzemplarzy różnych typów Pet. Idealnie nadaje się do tego kontener Map, w którym rolę kluczy będą pełnić nazwy klas Pet, a rolę wartości — liczniki egzemplarzy w postaci obiektów klasy Integer. Można więc będzie zapytać: „Ile mamy obiektów Hamster?". Do zliczania możemy wykorzystać słowo instanceof.

```
//: typeinfo/PetCount.java
// Użycie instanceof.
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class PetCount {
  static class PetCounter extends HashMap<String,Integer> {
    public void count(String type) {
      Integer quantity = get(type);
      if(quantity == null)
        put(type, 1);
      else
        put(type, quantity + 1);
    }
  }
  public static void
  countPets(PetCreator creator) {
    PetCounter counter= new PetCounter();
    for(Pet pet : creator.createArray(20)) {
      // Lista poszczególnych zwierzaków:
      printnb(pet.getClass().getSimpleName() + " ");
      if(pet instanceof Pet)
        counter.count("Pet");
      if(pet instanceof Dog)
        counter.count("Dog");
      if(pet instanceof Mutt)
        counter.count("Mutt");
      if(pet instanceof Pug)
        counter.count("Pug");
      if(pet instanceof Cat)
        counter.count("Cat");
      if(pet instanceof Manx)
        counter.count("EgyptianMau");
      if(pet instanceof Manx)
        counter.count("Manx");
      if(pet instanceof Manx)
        counter.count("Cymric");
      if(pet instanceof Rodent)
        counter.count("Rodent");
      if(pet instanceof Rat)
        counter.count("Rat");
      if(pet instanceof Mouse)
        counter.count("Mouse");
      if(pet instanceof Hamster)
        counter.count("Hamster");
    }
    // Wypisanie liczników:
    print();
    print(counter);
  }
  public static void main(String[] args) {
    countPets(new ForNameCreator());
  }
} /* Output:
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat EgyptianMau Hamster EgyptianMau Mutt Mutt
Cymric Mouse Pug Mouse Cymric
{Pug=3, Cat=9, Hamster=1, Cymric=7, Mouse=2, Mutt=3, Rodent=5, Pet=20, Manx=7, EgyptianMau=7,
Dog=6, Rat=2}
*///:~
```

W metodzie countPets() dochodzi do wypełnienia tablicy egzemplarzami losowanych podtypów Pet tworzonych za pomocą klasy PetCreator. Każdy z egzemplarzy Pet z tablicy jest następnie analizowany i zliczany przy użyciu słowa kluczowego instanceof.

Istnieje dosyć poważne ograniczenie możliwości operatora instanceof — można go używać wyłącznie do porównywania typów nazwanych, a nie obiektów Class. Patrząc na powyższy kod, można było odnieść wrażenie, że pisanie tych wszystkich wyrażeń instanceof jest bardzo nudne — niewątpliwie to prawda. Istnieje jednak inny sposób pozwalający na sprytną automatyzację wykorzystania operatora instanceof. Polega on na stworzeniu tablicy obiektów Class i użyciu ich podczas porównywania (bądź czujny — istnieje alternatywa także dla tego rozwiązania). Nie jest to tak duże ograniczenie, jak można by sądzić, gdyż pisząc tak wiele wyrażeń instanceof, na pewno w końcu doszedłbyś do wniosku, że projekt programu jest wadliwy.

Użycie literałów klas

Gdybyśmy zaimplementowali klasę PetCreator z zastosowaniem literałów klas, wynik byłby pod wieloma względami bardziej przejrzysty:

```
//: typeinfo/pets/LiteralPetCreator.java
// Użycie literałów klas.
package typeinfo.pets;
import java.util.*;

public class LiteralPetCreator extends PetCreator {
  // Bez bloku try.
  @SuppressWarnings("unchecked")
  public static final List<Class<? extends Pet>> allTypes =
    Collections.unmodifiableList(Arrays.<Class<? extends Pet>>asList(
      Pet.class, Dog.class, Cat.class,  Rodent.class,
      Mutt.class, Pug.class, EgyptianMau.class, Manx.class,
      Cymric.class, Rat.class, Mouse.class,Hamster.class));
  // Typy do losowania:
  private static final List<Class<? extends Pet>> types =
    allTypes.subList(allTypes.indexOf(Mutt.class),
      allTypes.size());
  public List<Class<? extends Pet>> types() {
    return types;
  }
  public static void main(String[] args) {
    System.out.println(types);
  }
} /* Output:
[class typeinfo.pets.Mutt, class typeinfo.pets.Pug, class typeinfo.pets.EgyptianMau, class
typeinfo.pets.Manx, class typeinfo.pets.Cymric, class typeinfo.pets.Rat, class typeinfo.pets.Mouse, class
typeinfo.pets.Hamster]
*///:~
```

W przyszłym przykładzie *PetCount3.java* będziemy musieli wstępnie załadować kontener Map z kompletem typów Pet (nie tylko z tymi, które mają być wybierane do losowania), stąd konieczność zastosowania listy allTypes. Lista types jest z kolei fragmentem allTypes (utworzonym wywołaniem List.subList()), obejmującym konkretne typy zwierzaków wykorzystywane już do losowego tworzenia obiektów Pet.

Tym razem stworzenie petTypes nie wymaga otoczenia blokiem try, ponieważ jest to przetwarzane w czasie kompilacji i dlatego nie zgłosi żadnego wyjątku w przeciwieństwie do Class.forName().

Dysponujemy teraz w bibliotece typeinfo.pets dwiema implementacjami PetCreator. Aby wybrać tę drugą jako implementację domyślną, możemy utworzyć *fasadę* wykorzystującą LiteralPetCreator:

```
// : typeinfo/pets/Pets.java
// Fasada wytwarzająca domyślną implementację PetCreator.
package typeinfo.pets;
import java.util.*;

public class Pets {
  public static final PetCreator creator =
    new LiteralPetCreator();
  public static Pet randomPet() {
    return creator.randomPet();
  }
  public static Pet[] createArray(int size) {
    return creator.createArray(size);
  }
  public static ArrayList<Pet> arrayList(int size) {
    return creator.arrayList(size);
  }
} ///:~
```

W ten sposób udostępniliśmy też delegacje metod randomPet(), createArray() i arrayList().

Ponieważ metoda PetCount.countPets() przyjmuje argument typu PetCreator, możemy łatwo przetestować implementację LiteralPetCreator (za pośrednictwem powyższej fasady):

```
// : typeinfo/PetCount2.java
import typeinfo.pets.*;

public class PetCount2 {
  public static void main(String[] args) {
    PetCount.countPets(Pets.creator);
  }
} /* (Execute to see output) *///:~
```

Wyjście programu jest identyczne jak programu *PetCount.java*.

Dynamiczne instanceof

Metoda isInstance() klasy Class pozwala na dynamiczne testowanie typu obiektu. Dzięki temu wszystkie żmudne wyrażenia instanceof z przykładu PetCount można usunąć:

```
// : typeinfo/PetCount3.java
// Użycie isInstance()
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
```

```
public class PetCount3 {
  static class PetCounter
  extends LinkedHashMap<Class<? extends Pet>,Integer> {
    public PetCounter() {
      super(MapData.map(LiteralPetCreator.allTypes, 0));
    }
    public void count(Pet pet) {
      // Class.isInstance() eliminuje stosowanie instanceof:
      for(Map.Entry<Class<? extends Pet>,Integer> pair
          : entrySet())
        if(pair.getKey().isInstance(pet))
          put(pair.getKey(), pair.getValue() + 1);
    }
    public String toString() {
      StringBuilder result = new StringBuilder("{");
      for(Map.Entry<Class<? extends Pet>,Integer> pair
          : entrySet()) {
        result.append(pair.getKey().getSimpleName());
        result.append("=");
        result.append(pair.getValue());
        result.append(", ");
      }
      result.delete(result.length()-2, result.length());
      result.append("}");
      return result.toString();
    }
  }
  public static void main(String[] args) {
    PetCounter petCount = new PetCounter();
    for(Pet pet : Pets.createArray(20)) {
      printnb(pet.getClass().getSimpleName() + " ");
      petCount.count(pet);
    }
    print();
    print(petCount);
  }
} /* Output:
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat EgyptianMau Hamster EgyptianMau Mutt Mutt
Cymric Mouse Pug Mouse Cymric
{Pet=20, Dog=6, Cat=9, Rodent=5, Mutt=3, Pug=3, EgyptianMau=2, Manx=7, Cymric=5, Rat=2,
Mouse=2, Hamster=1}
*///:~
```

Aby zliczyć wystąpienia różnych podtypów `Pet`, ładujemy do kontenera `Map` typy z `LiteralPetCreator.allTypes`. Korzystamy tutaj z klasy `net.mindview.util.MapData`, która przyjmuje implementację interfejsu `Iterable` (tu jest nią lista `allTypes`) oraz stałą (tu zero) i wypełnia kontener `Map` otrzymanymi (z `allTypes`) i wartościami (zerami). Bez wstępnego załadowania kontenera skończylibyśmy, zliczając jedynie typy wybrane do losowego generowania, bez typów bazowych jak `Pet` czy `Cat`.

Jak widać, metoda `isInstance()` wyeliminowała potrzebę wpisywania wyrażeń `instanceof`. W dodatku oznacza to, że można dodać nowe typy zwierząt — po prostu zmieniając zawartość tablicy `LiteralPetCreator.types`; reszta programu nie wymaga żadnych zmian (w przeciwieństwie do zastosowania wyrażeń `instanceof`).

Metoda `toString()` została przeciążona tak, aby ułatwić odczytywanie wyników podawanych na wyjście, ale tak, aby układ danych na wyjściu przypominał układ charakterystyczny dla wypisywania kontenera `Map`.

Zliczanie rekurencyjne

Kontener Map z PetCount3.PetCounter był wstępnie ładowany wszystkimi znanymi klasami hierarchii Pet. Zamiast operacji ładowania kontenera moglibyśmy użyć metody Class.isAssignableFrom() i utworzyć uniwersalne narzędzie, nieograniczone bynajmniej do zliczania typów Pet:

```
//: net/mindview/util/TypeCounter.java
// Zlicza typy w rodzinie typów.
package net.mindview.util;
import java.util.*;

public class TypeCounter extends HashMap<Class<?>,Integer>{
  private Class<?> baseType;
  public TypeCounter(Class<?> baseType) {
    this.baseType = baseType;
  }
  public void count(Object obj) {
    Class<?> type = obj.getClass();
    if(!baseType.isAssignableFrom(type))
      throw new RuntimeException(obj + " niewłaściwy typ: "
        + type + ", oczekiwany typ albo podtyp "
        + baseType);
    countClass(type);
  }
  private void countClass(Class<?> type) {
    Integer quantity = get(type);
    put(type, quantity == null ? 1 : quantity + 1);
    Class<?> superClass = type.getSuperclass();
    if(superClass != null &&
       baseType.isAssignableFrom(superClass))
      countClass(superClass);
  }
  public String toString() {
    StringBuilder result = new StringBuilder("{");
    for(Map.Entry<Class<?>,Integer> pair : entrySet()) {
      result.append(pair.getKey().getSimpleName());
      result.append("=");
      result.append(pair.getValue());
      result.append(", ");
    }
    result.delete(result.length()-2, result.length());
    result.append("}");
    return result.toString();
  }
} ///:~
```

Metoda count() przyjmuje w wywołaniu argument typu Class i wykorzystuje jego metodę isAssignableFrom() do wykonania dynamicznego testu przynależności przekazanego obiektu do danej hierarchii. Metoda countClass() najpierw zlicza dokładny typ klasy. Potem, jeśli baseType daje się przypisywać z nadklasy, następuje rekurencyjne wywołanie countClass() dla nadklasy.

```
//: typeinfo/PetCount4.java
import typeinfo.pets.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
```

```
public class PetCount4 {
  public static void main(String[] args) {
    TypeCounter counter = new TypeCounter(Pet.class);
    for(Pet pet : Pets.createArray(20)) {
      printnb(pet.getClass().getSimpleName() + " ");
      counter.count(pet);
    }
    print();
    print(counter);
  }
} /* Output: (Sample)
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat EgyptianMau Hamster EgyptianMau Mutt Mutt
Cymric Mouse Pug Mouse Cymric
{Mouse=2, Dog=6, Manx=7, EgyptianMau=2, Rodent=5, Pug=3, Mutt=3, Cymric=5, Cat=9, Hamster=1,
Pet=20, Rat=2}
*///:~
```

Jak widać, zliczone zostały zarówno typy bazowe, jak i typy konkretne.

Ćwiczenie 11. Dodaj do biblioteki *typeinfo.pets* klasę `Gerbil` (i zmień wszystkie przykłady w tym rozdziale tak, aby obejmowały nową klasę (2).

Ćwiczenie 12. Użyj klasy `TypeCounter` z klasą *CoffeeGenerator.java* z rozdziału „Typy ogólne" (3).

Ćwiczenie 13. Użyj klasy `TypeCounter` z klasą *RegisteredFactories.java* z tego rozdziału (3).

Wytwórnie rejestrowane

Generowanie obiektów z hierarchii `Pet` jest o tyle utrudnione, że za każdym razem, kiedy uzupełnimy hierarchię o nowy podtyp, musimy pamiętać o stosownym uzupełnieniu w *LiteralPetCreator.java*. W systemie, w którym regularnie hierarchia byłaby rozbudowywana, stanowiłoby to znaczną uciążliwość.

Można by pomyśleć o dodaniu do każdej podklasy statycznego inicjalizatora, tak aby ów inicjalizator dodawał daną klasę do jakiejś zewnętrznej listy. Niestety, statyczny inicjalizator jest wywoływany jedynie przy ładowaniu klasy, co prowadzi do problemu jajka i kury — generator nie posiada klasy na liście, więc nie utworzy obiektu tej klasy, przez co klasa nie zostanie załadowana, a więc nie trafi na listę.

Zasadniczo jesteśmy zmuszeni do samodzielnego, ręcznego tworzenia listy (chyba że pokusimy się o oprogramowanie narzędzia przeszukującego kod źródłowy i na jego podstawie tworzącego i kompilującego stosowną listę). Najlepsze, co można zrobić, to umieścić listę w centralnym, „dobrze widocznym" miejscu. Najlepszym miejscem byłaby zapewne bazowa klasa hierarchii.

Następna zmiana polegałaby na przeniesieniu tworzenia obiektu do samej klasy, a to za sprawą wzorca projektowego *Factory Method*. Metoda wytwórcza może być wywoływana polimorficznie i tworzyć dla użytkownika obiekt odpowiedniego typu. W bardzo

uproszczonej wersji byłaby to metoda podobna do metody create() poniższego interfejsu Factory:

```
//: typeinfo/factory/Factory.java
package typeinfo.factory;
public interface Factory<T> { T create(); } ///:~
```

Parametr ogólny T pozwala metodzie create() zwracać obiekty różnych typów, zależnych od implementacji interfejsu Factory. Mamy tu do czynienia z kowariancją typów zwracanych.

W poniższym przykładzie klasa bazowa Part zawiera listę obiektów-wytwórni. Wytwórnie dla typów, które mają być wytwarzane za pomocą metody createRandom(), są „rejestrowane" w klasie bazowej; rejestracja polega na dodaniu wytwórni do listy partFactories:

```
//: typeinfo/RegisteredFactories.java
// Rejestrowanie wytwórni klas w klasie bazowej.
import typeinfo.factory.*;
import java.util.*;

class Part {
  public String toString() {
    return getClass().getSimpleName();
  }
  static List<Factory<? extends Part>> partFactories =
    new ArrayList<Factory<? extends Part>>();
  static {
    // Metoda Collections.addAll() powoduje ostrzeżenie
    // "unchecked generic array creation ... for varargs parameter".
    partFactories.add(new FuelFilter.Factory());
    partFactories.add(new AirFilter.Factory());
    partFactories.add(new CabinAirFilter.Factory());
    partFactories.add(new OilFilter.Factory());
    partFactories.add(new FanBelt.Factory());
    partFactories.add(new PowerSteeringBelt.Factory());
    partFactories.add(new GeneratorBelt.Factory());
  }
  private static Random rand = new Random(47);
  public static Part createRandom() {
    int n = rand.nextInt(partFactories.size());
    return partFactories.get(n).create();
  }
}

class Filter extends Part {}

class FuelFilter extends Filter {
  // Utworzenie wytwórni dla każdego konkretnego typu:
  public static class Factory
  implements typeinfo.factory.Factory<FuelFilter> {
    public FuelFilter create() { return new FuelFilter(); }
  }
}

class AirFilter extends Filter {
  public static class Factory
```

```
    implements typeinfo.factory.Factory<AirFilter> {
      public AirFilter create() { return new AirFilter(); }
    }
}

class CabinAirFilter extends Filter {
  public static class Factory
  implements typeinfo.factory.Factory<CabinAirFilter> {
    public CabinAirFilter create() {
      return new CabinAirFilter();
    }
  }
}

class OilFilter extends Filter {
  public static class Factory
  implements typeinfo.factory.Factory<OilFilter> {
    public OilFilter create() { return new OilFilter(); }
  }
}

class Belt extends Part {}

class FanBelt extends Belt {
  public static class Factory
  implements typeinfo.factory.Factory<FanBelt> {
    public FanBelt create() { return new FanBelt(); }
  }
}

class GeneratorBelt extends Belt {
  public static class Factory
  implements typeinfo.factory.Factory<GeneratorBelt> {
    public GeneratorBelt create() {
      return new GeneratorBelt();
    }
  }
}

class PowerSteeringBelt extends Belt {
  public static class Factory
  implements typeinfo.factory.Factory<PowerSteeringBelt> {
    public PowerSteeringBelt create() {
      return new PowerSteeringBelt();
    }
  }
}

public class RegisteredFactories {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      System.out.println(Part.createRandom());
  }
} /* Output:
GeneratorBelt
CabinAirFilter
GeneratorBelt
AirFilter
```

```
PowerSteeringBelt
CabinAirFilter
FuelFilter
PowerSteeringBelt
PowerSteeringBelt
FuelFilter
*///:~
```

Nie wszystkie klasy hierarchii nadają się do konkretyzacji (do tworzenia egzemplarzy); tutaj `Filter` i `Belt` to jedynie klasyfikatory, których egzemplarze nie mają sensu — tworzymy tylko obiekty ich klas pochodnych. Jeśli natomiast klasa *ma* podlegać wytwarzaniu za pośrednictwem `createRandom()`, powinna zawierać wewnętrzną klasę `Factory`. Jedynym sposobem innego wykorzystania nazwy `Factory` jest kwalifikacja `typeinfo.factory.Factory`.

Choć do dodawania wytwórni do listy moglibyśmy wykorzystać metodę `Collections.addAll()`, kompilator wyraziłby wtedy swoje niezadowolenie, ostrzegając o „utworzeniu tablicy uogólnionej" (choć to niby niemożliwe, o czym przekonamy się w rozdzialc „Typy ogólne"), dlatego postanowiłem uciec się do wywołania `add()`. Metoda `createRandom()` wybiera losowo obiekt wytwórni z kontenera `partFactories` i wywołuje na rzecz tego obiektu metodę `create()` zwracającą nowy egzemplarz `Part`.

Ćwiczenie 14. Konstruktor jest swego rodzaju metodą wytwórczą. Zmodyfikuj plik *RegisteredFactories.java* tak, aby zamiast używać jawnej metody wytwórczej, na liście przechowywane były obiekty `Class`, a nowe egzemplarze tworzone były wywołaniami `newInstance()` (4).

Ćwiczenie 15. Zaimplementuj nową wersję `PetCreator` na bazie rejestrowanych wytwórni; zmodyfikuj fasadę `Pets` tak, aby wykorzystywała nowy model tworzenia obiektów zamiast poprzednich dwóch. Upewnij się, że reszta przykładów używających kodu *Pets.java* wciąż działa poprawnie (4).

Ćwiczenie 16. Zmodyfikuj hierarchię `Coffee` z rozdziału „Typy ogólne" tak, aby wykorzystywała technikę rejestrowania wytwórni (4).

instanceof a równoważność obiektów Class

Pytając o informacje na temat typu, mamy do czynienia z istotną różnicą pomiędzy obiema postaciami `instanceof` (czyli `instanceof` lub `isInstance()`, które dają równoważny wynik) a bezpośrednim porównaniem obiektów typu `Class`. Oto przykład, który ją pokazuje:

```
//: typeinfo/FamilyVsExactType.java
// Różnica pomiędzy przynależnością a równoważnością
package typeinfo;
import static net.mindview.util.Print.*;

class Base {}
class Derived extends Base {}
```

```
public class FamilyVsExactType {
  static void test(Object x) {
    print("Testowanie x typu " + x.getClass());
    print("x instanceof Base " + (x instanceof Base));
    print("x instanceof Derived "+ (x instanceof Derived));
    print("Base.isInstance(x) "+ Base.class.isInstance(x));
    print("Derived.isInstance(x) " +
      Derived.class.isInstance(x));
    print("x.getClass() == Base.class " +
      (x.getClass() == Base.class));
    print("x.getClass() == Derived.class " +
      (x.getClass() == Derived.class));
    print("x.getClass().equals(Base.class)) "+
      (x.getClass().equals(Base.class)));
    print("x.getClass().equals(Derived.class)) " +
      (x.getClass().equals(Derived.class)));
  }
  public static void main(String[] args) {
    test(new Base());
    test(new Derived());
  }
} /* Output:
Testowanie x typu class typeinfo.Base
x instanceof Base true
x instanceof Derived false
Base.isInstance(x) true
Derived.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derived.class false
x.getClass().equals(Base.class)) true
x.getClass().equals(Derived.class)) false
Testowanie x typu class typeinfo.Derived
x instanceof Base true
x instanceof Derived true
Base.isInstance(x) true
Derived.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derived.class true
x.getClass().equals(Base.class)) false
x.getClass().equals(Derived.class)) true
*///:~
```

Metoda `test()` sprawdza typ argumentów, stosując obie postacie `instanceof`. Następnie pobiera referencję do `Class` i stosuje porównanie `==` oraz `equals()` do sprawdzenia równoważności obiektów `Class`. Na szczęście `instanceof` i `isInstance()` dają dokładnie te same wyniki, podobnie jak `equals()` i `==`. Ale po wykonaniu samych testów nasuwają odmienne wnioski. W przypadku pojęcia typu `instanceof` pyta: „Czy jesteś tą klasą lub klasą z niej się wywodzącą?". Z drugiej strony, jeżeli porównać obiekty `Class`, stosując `==`, to porównanie nie uwzględnia dziedziczenia — albo jest to dokładnie ten sam typ, albo nie.

Refleksja — informacja o klasie w czasie wykonania

Jeżeli nie znamy dokładnego typu obiektu, to RTTI może nam podpowiedzieć. Istnieje jednak pewne ograniczenie: typ musi być znany w czasie kompilacji, aby można było go wykryć, stosując RTTI, i wykorzystać tę informację. Upraszczając, kompilator musi wiedzieć o wszystkich klasach, z którymi pracujemy.

Na pierwszy rzut oka nie wygląda to na zbyt poważne ograniczenie, ale przypuśćmy, że dostaliśmy referencję do obiektu spoza przestrzeni naszego programu. Tak naprawdę klasa takiego obiektu nie jest nawet dostępna dla programu podczas kompilacji. Załóżmy na przykład, że dostaliśmy blok bajtów z pliku lub poprzez połączenie sieciowe i powiedziano nam, że reprezentują one jakąś klasę. Ponieważ kompilator nie może wiedzieć o klasie podczas kompilacji kodu, w jaki sposób może użyć takiej klasy?

Wydaje się, że w tradycyjnym środowisku programowania ten scenariusz zachodzi bardzo rzadko. Ale w miarę przechodzenia do programowania na większą skalę pojawiają się ważne sytuacje, w których ma to miejsce. Pierwszym jest programowanie bazujące na komponentach, podczas którego buduje się projekty, stosując *Rapid Application Development* (RAD) w narzędziach do budowy aplikacji określanych mianem *Integrated Development Environment* albo w skrócie IDE. Jest to wizualne rozwiązanie tworzenia oprogramowania poprzez przenoszenie ikon reprezentujących komponenty na formatkę. Te komponenty są następnie konfigurowane poprzez ustawienie kilku z ich parametrów w czasie programowania. Taka konfiguracja w czasie projektowania wymaga, aby każdy komponent udostępniał swoje właściwości i by pozwalał na odczyt oraz modyfikowanie ich wartości. Poza tym komponenty, które obsługują zdarzenia GUI, muszą eksponować informacje o stosownych metodach, tak by środowisko IDE mogło pomagać programiście w przesłanianiu takich metod obsługi zdarzeń. Refleksja dostarcza mechanizm detekcji dostępnych metod i podaje nazwy tych metod. Java umożliwia konstrukcję programowania bazującego na komponentach poprzez standard JavaBeans (opisany w rozdziale „Graficzne interfejsy użytkownika").

Inną ważną motywacją do odkrywania informacji o klasie w czasie działania programu jest zapewnienie zdolności tworzenia i wykonania obiektów na oddalonych platformach poprzez sieć. Nazywa się to *Remote Method Invocation* (RMI) i pozwala programom Javy na posiadanie obiektów rozproszonych na wielu maszynach. Takie rozproszenie może mieć miejsce z wielu powodów. Przypuśćmy, na przykład, że wykonujemy zadanie wymagające intensywnych obliczeń i chcemy go podzielić, by umieścić części na maszynach, które nie są zajęte, aby przyspieszyć cały proces. W pewnych sytuacjach można chcieć zamieścić kod, który dotyczy konkretnego typu zadań (np. „reguły biznesowe" w przypadku wielowarstwowej architektury klient-serwer) na konkretnej maszynie tak, aby stała się ona wspólnym repozytorium definicji pewnych działań. Definicje takie można łatwo zmieniać, a zmiana propagowana jest w całym systemie (jest to ciekawy pomysł, gdyż maszyna ta istnieje wyłącznie po to, by umożliwić łatwiejsze zmiany oprogramowania!). Programowanie rozproszone wspiera również specjalizowany sprzęt, co może być przydatne w szczególnych zadaniach — przykładowo znalezienia macierzy odwrotnej — ale niewłaściwe lub zbyt kosztowne dla tradycyjnego programowania.

Klasa `Class` (opisana wcześniej w tym rozdziale) obsługuje pojęcie *refleksji* (ang. *reflection*), ale mamy też dodatkową bibliotekę `java.lang.reflect`, która zawiera klasy `Field`, `Method` oraz `Constructor` (każda z nich implementuje interfejs `Member`). Obiekty tych typów tworzone są przez JVM w czasie wykonania, by reprezentować odpowiadające składowe w nieznanej klasie. Można stosować `Constructor` do tworzenia nowych obiektów, metody `get()` i `set()` do odczytu i modyfikacji pól powiązanych z obiektami `Field` oraz metodę `invoke()` do wywołania metody związanej z obiektem `Method`. W dodatku można użyć wygodnych metod: `getFields()`, `getMethods()`, `getConstructors()` i innych, aby uzyskać tablicę obiektów reprezentujących odpowiednio: pola, metody i konstruktory (dowiesz się więcej, wyszukując opis klasy `Class` w dokumentacji Javadoc). Zatem informacje o klasie dla jakichś anonimowych obiektów mogą być całkowicie ustalone w czasie działania i nic nie trzeba wiedzieć podczas kompilacji.

Istotne jest, aby zdać sobie sprawę z tego, że nie ma żadnej magii w mechanizmie refleksji. Stosując go do współdziałania z obiektem nieznanego typu, JVM zwyczajnie podejrzy obiekt i zobaczy, że należy on do określonej klasy (podobnie jak RTTI), ale później, zanim może zrobić cokolwiek innego, musi załadowany obiekt `Class`. W ten sposób plik *.class* wybranego typu musi być dostępny dla JVM — albo lokalnie na maszynie, albo poprzez sieć. Tak więc prawdziwa różnica pomiędzy RTTI a refleksją polega na tym, że w przypadku RTTI kompilator otwiera i analizuje plik *.class* w czasie kompilacji. Innymi słowy, można wywołać wszystkie metody obiektu w „normalny" sposób. W przypadku refleksji plik ten nie jest dostępny w czasie kompilacji, a jest otwierany i sprawdzany w środowisku uruchomieniowym.

Ekstraktor metod

Rzadko będziesz miał okazję używać refleksji bezpośrednio; przydadzą się jednak do tworzenia bardziej dynamicznego kodu. Mechanizm refleksji służy zasadniczo do obsługi innych właściwości Javy, np. serializacji obiektów oraz komponentów JavaBeans (zobacz kolejne rozdziały). Bywa jednak, że możliwość dostania się w sposób dynamiczny do informacji o klasie za pomocą refleksji jest wybawieniem.

Rozważmy przykład ekstraktora metod klasy. Kod źródłowy lub dokumentacja Javadoc pokazują tylko metody, które są zdefiniowane lub przesłonięte *w ramach definicji danej klasy*. Może jednak istnieć wiele więcej dostępnych metod, które pochodzą z klas bazowych. Ich wykrycie jest zarówno nużące, jak i czasochłonne[2]. Na szczęście mechanizm refleksji zapewnia sposób na napisanie prostego narzędzia, które będzie automatycznie pokazywać cały interfejs klasy. Oto w jaki sposób działa:

```
// : typeinfo/ShowMethods.java
// Zastosowanie refleksji do ujawnienia kompletu metod
// klasy, również tych pochodzących z klasy bazowej.
// {Args: ShowMethods}
import java.lang.reflect.*;
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class ShowMethods {
```

[2] Szczególnie w przeszłości. Firma Sun znacznie poprawiła swą HTML-ową dokumentację Javy, więc teraz łatwiej jest przejrzeć metody klasy bazowej.

```
  private static String usage =
    "Stosowanie:\n" +
    "ShowMethods kwalifikowana.nazwa.klasy\n" +
    "Aby wypisać wszystkie metody klasy albo:\n" +
    "ShowMethods kwalifikowana.nazwa.klasy słowo\n" +
    "Aby wyszukać metody ze słowem 'słowo'";
  private static Pattern p = Pattern.compile("\\w+\\.");
  public static void main(String[] args) {
    if(args.length < 1) {
      print(usage);
      System.exit(0);
    }
    int lines = 0;
    try {
      Class<?> c = Class.forName(args[0]);
      Method[] methods = c.getMethods();
      Constructor[] ctors = c.getConstructors();
      if(args.length == 1) {
        for(Method method : methods)
          print(
            p.matcher(method.toString()).replaceAll(""));
        for(Constructor ctor : ctors)
          print(p.matcher(ctor.toString()).replaceAll(""));
        lines = methods.length + ctors.length;
      } else {
        for(Method method : methods)
          if(method.toString().indexOf(args[1]) != -1) {
            print(
              p.matcher(method.toString()).replaceAll(""));
            lines++;
          }
        for(Constructor ctor : ctors)
          if(ctor.toString().indexOf(args[1]) != -1) {
            print(p.matcher(
              ctor.toString()).replaceAll(""));
            lines++;
          }
      }
    } catch(ClassNotFoundException e) {
      print("Brak klasy: " + e);
    }
  }
} /* Output:
public static void main(String[])
public native int hashCode()
public final native Class getClass()
public final void wait(long,int) throws InterruptedException
public final void wait() throws InterruptedException
public final native void wait(long) throws InterruptedException
public boolean equals(Object)
public String toString()
public final native void notify()
public final native void notifyAll()
public ShowMethods()
*///:~
```

Metody `getMethods()` i `getConstructors()` z klasy `Class` zwracają odpowiednio tablicę obiektów `Method` oraz `Constructor`. Każda z tych klas posiada kolejne metody, by można było rozłożyć na czynniki nazwy argumentów i wartości zwracane metod, które

reprezentują. Można jednak po prostu użyć wywołania `toString()`, jak to ma miejsce w przykładzie, aby uzyskać ciąg znaków z pełną sygnaturą metody. Reszta kodu to już zwyczajne pozyskanie informacji z wiersza poleceń, sprawdzenie, czy konkretna sygnatura odpowiada łańcuchowi docelowemu (dzięki `indexOf()`), i obcięcie kwalifikatorów nazw przy użyciu wyrażeń regularnych (omawianych w rozdziale „Ciągi znaków").

Rezultat otrzymywany z `Class.forName()` nie może być znany podczas kompilacji i dlatego cała informacja o sygnaturze metody jest pobierana w czasie działania programu. Jeżeli prześledzisz dokumentację dotyczącą refleksji, to zauważysz, że istnieje wystarczające wsparcie, aby właściwie przygotować i wykonać metodę dowolnego obiektu, który jest zupełnie nieznany w czasie kompilacji (przykłady przedstawione zostaną później). Choć początkowo można przypuszczać, że możliwości te nigdy nie okażą się przydatne, to jednak pełna wartość refleksji jest bardzo zaskakująca.

Interesującym eksperymentem jest wywołanie:

```
java ShowMethods ShowMethods
```

Daje ono wykaz, który zawiera domyślny konstruktor publiczny, nawet jeżeli żaden konstruktor nie został zdefiniowany. Konstruktor, który widać, jest jedynym automatycznie tworzonym przez kompilator. Jeśli później zmienimy klasę `ShowMethods` na nie `public` (czyli pakietową), to dodany konstruktor nie będzie więcej prezentowany. Generowany konstruktor domyślny jest bowiem automatycznie tworzony z tym samym dostępem co jego klasa.

Kolejnym ciekawym eksperymentem jest wywołanie programu `java ShowMethods java.lang.String` z dodatkowym argumentem `char`, `int`, `String` itp.

Narzędzie to może naprawdę przyczynić się do oszczędności czasu podczas programowania, gdy nie pamiętamy, czy klasa posiada jakąś metodę, i nie chce nam się przeglądać hierarchii klas zamieszczonej w dokumentacji, albo jeżeli nie wiemy, czy klasa może coś zrobić, powiedzmy, z obiektami `Color`.

Rozdział „Graficzne interfejsy użytkownika" zawiera wersję tego programu z graficznym interfejsem użytkownika (przystosowaną do uzyskiwania informacji dla komponentów Swing), tak więc możesz sobie go uruchomić podczas tworzenia kodu, aby mieć możliwość obejrzenia go.

Ćwiczenie 17. Zmodyfikuj wyrażenie regularne w programie *ShowMethods.java* tak, aby dodatkowo usuwane były także słowa kluczowe `native` oraz `final` (podpowiedź: zastosuj operator alternatywy — `|`) (2).

Ćwiczenie 18. Uczyń klasę `ShowMethods` klasą niepubliczną (z dostępem pakietowym) i sprawdź, czy generowany konstruktor domyślny faktycznie zniknął z listingu (1).

Ćwiczenie 19. W pliku *ToyTest.java* skorzystaj z refleksji w celu utworzenia obiektu `Toy` za pomocą konstruktora innego niż domyślny (4).

Ćwiczenie 20. Przejrzyj (pod adresem *http://java.sun.com*) dokumentację JDK interfejsu `java.lang.Class`. Napisz program, który będzie wywoływany z argumentem w postaci nazwy klasy, a potem wykorzysta metody `Class` do wypisania wszystkich dostępnych mu informacji o tej klasie. Wypróbuj program z wybraną klasą biblioteki standardowej i klasą własną (5).

Dynamiczne proxy

Proxy to jeden z podstawowych wzorców projektowych. To obiekt, który jest wstawiany w miejsce „prawdziwego" obiektu w celu udostępniania innych albo dodatkowych operacji — zwykle angażujących komunikację z owym „prawdziwym" obiektem — proxy występuje więc jako pośrednik albo brama. Oto prosty przykład ilustrujący strukturę proxy:

```
//: typeinfo/SimpleProxyDemo.java
import static net.mindview.util.Print.*;

interface Interface {
  void doSomething();
  void somethingElse(String arg);
}

class RealObject implements Interface {
  public void doSomething() { print("doSomething"); }
  public void somethingElse(String arg) {
    print("somethingElse " + arg);
  }
}

class SimpleProxy implements Interface {
  private Interface proxied;
  public SimpleProxy(Interface proxied) {
    this.proxied = proxied;
  }
  public void doSomething() {
    print("SimpleProxy doSomething");
    proxied.doSomething();
  }
  public void somethingElse(String arg) {
    print("SimpleProxy somethingElse " + arg);
    proxied.somethingElse(arg);
  }
}

class SimpleProxyDemo {
  public static void consumer(Interface iface) {
    iface.doSomething();
    iface.somethingElse("bonobo");
  }
  public static void main(String[] args) {
    consumer(new RealObject());
    consumer(new SimpleProxy(new RealObject()));
  }
} /* Output:
doSomething
somethingElse bonobo
SimpleProxy doSomething
doSomething
SimpleProxy somethingElse bonobo
somethingElse bonobo
*///:~
```

Ponieważ metoda `consumer()` oczekuje przekazania implementacji interfejsu `Interface`, nie wie, że zamiast obiektu `RealObject` otrzymuje do dyspozycji `Proxy` — metoda nie może ich rozróżnić, bo oba implementują interfejs `Interface`. Tymczasem `Proxy`, wstawiony pomiędzy klienta a obiekt docelowy `RealObject`, przyjmuje zlecenia operacji i realizuje je poprzez delegację wywołań do odpowiednich metod `RealObject`.

Proxy może się przydać wszędzie tam, gdzie trzeba odizolować dodatkowe operacje od obiektu „docelowego", zwłaszcza tam, gdzie chcielibyśmy móc łatwo wybierać pomiędzy realizacją owych dodatkowych operacji a ich zaniechaniem (zadaniem wzorców projektowych jest hermetyzowanie tego, co zmienne). Na przykład gdybyśmy zechcieli rejestrować wywołania metod `RealObject`, albo mierzyć narzuty związane z tymi wywołaniami, nie chcielibyśmy na trwałe osadzać podobnego kodu w aplikacji (wszak nie on stanowi jej sens i sedno) — proxy nadaje się do tego idealnie, bo jak łatwo je wstawić, tak samo łatwo można je usunąć.

Koncepcja *proxy dynamicznego* posuwa tę koncepcję o krok dalej, zakładając dynamiczne tworzenie obiektu proxy i dynamiczne obsługiwanie wywołań metod korzystających z takiego pośrednictwa. Wszystkie wywołania czynione na rzecz dynamicznego proxy są przekierowywane do pojedynczego *obiektu obsługi wywołań* (ang. *invocation handler*), którego zadaniem jest identyfikacja wywołania i odpowiednia reakcja. Oto przykład *SimpleProxyDemo.java*, przepisany z użyciem dynamicznego proxy:

```
//: typeinfo/SimpleDynamicProxy.java
import java.lang.reflect.*;

class DynamicProxyHandler implements InvocationHandler {
  private Object proxied;
  public DynamicProxyHandler(Object proxied) {
    this.proxied = proxied;
  }
  public Object
  invoke(Object proxy, Method method, Object[] args)
  throws Throwable {
    System.out.println("**** proxy: " + proxy.getClass() +
      ", metoda: " + method + ", argumenty: " + args);
    if(args != null)
      for(Object arg : args)
        System.out.println("  " + arg);
    return method.invoke(proxied, args);
  }
}

class SimpleDynamicProxy {
  public static void consumer(Interface iface) {
    iface.doSomething();
    iface.somethingElse("bonobo");
  }
  public static void main(String[] args) {
    RealObject real = new RealObject();
    consumer(real);
    // Wstawienie proxy i ponowne wywołanie:
    Interface proxy = (Interface)Proxy.newProxyInstance(
      Interface.class.getClassLoader(),
```

```
        new Class[]{ Interface.class },
        new DynamicProxyHandler(real));
    consumer(proxy);
  }
} /* Output: (95% match)
doSomething
somethingElse bonobo
**** proxy: klasa $Proxy0, metoda: public abstract void Interface.doSomething(), argumenty: null
doSomething
**** proxy: klasa $Proxy0, metoda: public abstract void Interface.somethingElse(java.lang.String),
argumenty: [Ljava.lang.Object;@42e816
 bonobo
somethingElse bonobo
*///:~
```

Proxy dynamiczne tworzy się wywołaniem statycznej metody `Proxy.newProxyInstance()`, która wymaga przekazania class loadera (zasadniczo można przekazać loader z obiektu, który już został załadowany) oraz listy interfejsów (nie klas ani klas abstrakcyjnych), które mają być implementowane przez proxy i implementację interfejsu `InvocationHandler`. Dynamiczne proxy będzie przekierowywało wszystkie wywołania do tejże implementacji, więc konstruktor obiektu obsługującego wywołania zwykle otrzymuje w wywołaniu referencję obiektu docelowego, tak aby mógł delegować do niego wywołania po wykonaniu swoich zadań dodatkowych.

Metoda `invoke()` otrzymuje w wywołaniu obiekt proxy, na wypadek gdyby trzeba było rozróżnić źródła wywołań — ale zazwyczaj jest nam wszystko jedno. Trzeba jednak zachować ostrożność przy wywoływaniu metod obiektu proxy w ramach metody `invoke()` — wywołania z interfejsu są wszak kierowane przez proxy.

Ogólnie rzecz biorąc, obsługa wywołania w proxy polega na wykonaniu owych dodatkowych czynności, a następnie wywołaniu metody `Method.invoke()` w celu propagacji wywołania do obiektu docelowego z niezbędnymi argumentami. Na pozór stanowi to ograniczenie, bo można tu wykonywać jedynie operacje ogólne, to jest niezależne od wywoływanej metody (jak np. rejestrowanie wywołań). Okazuje się jednak, że można filtrować wywołania metod i w ten sposób specjalizować działanie proxy:

```
//: typeinfo/SelectingMethods.java
// Wyszukiwanie konkretnych metod w dynamicznym proxy.
import java.lang.reflect.*;
import static net.mindview.util.Print.*;

class MethodSelector implements InvocationHandler {
  private Object proxied;
  public MethodSelector(Object proxied) {
    this.proxied = proxied;
  }
  public Object
  invoke(Object proxy, Method method, Object[] args)
  throws Throwable {
    if(method.getName().equals("interesting"))
      print("Proxy wykryło interesującą je metodę");
    return method.invoke(proxied, args);
  }
}
```

```
interface SomeMethods {
  void boring1();
  void boring2();
  void interesting(String arg);
  void boring3();
}

class Implementation implements SomeMethods {
  public void boring1() { print("boring1"); }
  public void boring2() { print("boring2"); }
  public void interesting(String arg) {
    print("interesująca. " + arg);
  }
  public void boring3() { print("boring3"); }
}

class SelectingMethods {
  public static void main(String[] args) {
    SomeMethods proxy= (SomeMethods)Proxy.newProxyInstance(
      SomeMethods.class.getClassLoader(),
      new Class[]{ SomeMethods.class },
      new MethodSelector(new Implementation()));
    proxy.boring1();
    proxy.boring2();
    proxy.interesting("bonobo");
    proxy.boring3();
  }
} /* Output:
boring1
boring2
Proxy wykryło interesującą je metodę
interesująca. bonobo
boring3
*///:~
```

Tu filtrowaliśmy wywołania pod kątem nazw metod, ale selektywny wybór wywołania metody do obsłużenia w proxy może opierać się również na innych elementach sygnatury; wywołania można filtrować nawet według wartości argumentów.

Dynamiczne proxy dla wywołań metod z pewnością nie jest narzędziem codziennego użytku, ale nie da się też mu odmówić przydatności do rozwiązywania szczególnej kategorii problemów. O wzorcu *Proxy* i innych wzorcach projektowych dowiesz się więcej z książki *Thinking in Patterns* (zobacz *www.MindView.net*) i klasycznej już publikacji *Design Patterns* autorstwa Ericha Gammy i spółki (Addison-Wesley, 1995).

Ćwiczenie 21. Zmień plik *SimpleProxyDemo.java* tak, aby proxy zajmowało się pomiarem czasu wywołania metod (3).

Ćwiczenie 22. Zmień plik *SimpleDynamicProxy.java* tak, aby proxy zajmowało się pomiarem czasu wywołania metod (3).

Ćwiczenie 23. W metodzie `invoke()` w *SimpleDynamicProxy.java* spróbuj wypisać argument proxy i wyjaśnij, co się wtedy dzieje (3).

Projekt[3]. Napisz system, który wykorzysta dynamiczne proxy do implementacji transakcji, gdzie proxy będą zajmować się zatwierdzaniem operacji w przypadku skutecznej realizacji wywołania (to znaczy braku wyjątków w wywołaniu propagowanym do obiektu docelowego) bądź wycofywania operacji w przypadku niepowodzenia. Zatwierdzanie i wycofywanie powinno operować na zewnętrznym pliku tekstowym pozostającym poza kontrolą wyjątków Javy. Zwróć szczególną uwagę na cechę *niepodzielności* transakcji.

Obiekty puste

Kiedy sygnalizujemy nieobecność obiektu wbudowaną wartością `null`, musimy przy każdorazowym odwołaniu do obiektu sprawdzać, czy referencja nie jest czasem pusta. To dość uciążliwe, a i ryzykowne, bowiem łatwo zapomnieć o koniecznym teście. Problem w tym, że jedynym własnym aspektem zachowania `null` jest zgłoszenie wyjątku `NullPointerException` w reakcji na jakiekolwiek operacje na referencji pustej. Niekiedy warto więc wprowadzić do programu pojęcie obiektu pustego[4] (*Null Object*), który akceptowałby komunikaty do obiektu, w imieniu którego występuje, ale zwracał wartości sygnalizujące brak adresata komunikatów. W ten sposób można przyjąć w programie, że wszystkie obiekty są poprawne i zaniechać uciążliwego sprawdzania referencji pustych.

Można sobie wyobrazić język programowania, który automatycznie tworzyłby dla programisty obiekty puste, w praktyce jednak nie ma sensu używać takich obiektów wszędzie — niekiedy wystarczy właśnie sprawdzanie pod kątem wartości `null`, niekiedy zaś można całkowicie bezpiecznie założyć, że referencja pusta w danym miejscu nie wystąpi, wreszcie niekiedy sygnalizacja za pośrednictwem wyjątku `NullPointerException` jest po prostu wygodna. Miejsce obiektów pustych widzę raczej „bliżej danych", a więc przy obiektach reprezentujących encje z dziedziny problemu. Prostym przykładem może być obecna w wielu modelach klasa `Person`. Zdarza się, że w kodzie nie mamy do dyspozycji obiektu `Person` (albo mamy, ale bez informacji o reprezentowanej nim osobie); tradycyjnie w takich sytuacjach zastosowalibyśmy referencję pustą (`null`) i w odwołaniach do obiektu klasy użylibyśmy stosownego testu. Możemy zamiast tego wykorzystać wzorzec obiektu pustego. Ale choć obiekt pusty będzie reagował na wszystkie komunikaty, które odebrałby także obiekt docelowy, musimy przecież mieć możliwość sprawdzenia, czy ten ostatni faktycznie istnieje. Najprościej wykorzystać do tego interfejs-znacznik:

```
//: net/mindview/util/Null.java
package net.mindview.util;
public interface Null {} ///:~
```

W ten sposób możemy wykrywać obiekty puste za pomocą słowa `instanceof`, a co ważniejsze, nie musimy dodawać do każdej ze swoich klas metody `isNull()` (która byłaby, mimo wszystko, jedynie alternatywnym sposobem pozyskania informacji RTTI — ale dlaczego nie skorzystać ze sposobów wbudowanych?):

[3] Proponowane projekty można wykorzystać na przykład jako warunki zaliczenia semestrów. Rozwiązania dostępne dla zwykłych ćwiczeń nie zawierają oczywiście rozwiązań projektów.

[4] Na pomysł ten wpadli Bobby Woolf i Bruce Anderson. Wzorzec ten można rozpatrywać jako szczególny przypadek wzorca *Strategy*. Wariantem wzorca *Null Object* jest wzorzec projektowy *Null Iterator*, który czyni iterację przez węzły gałęzi hierarchii operacją transparentną dla klienta (klient może tak samo przeglądać liście, jak i gałęzie).

```
//: typeinfo/Person.java
// Klasa z obiektem pustym.
import net.mindview.util.*;

class Person {
  public final String first;
  public final String last;
  public final String address;
  // itd.
  public Person(String first, String last, String address){
    this.first = first;
    this.last = last;
    this.address = address;
  }
  public String toString() {
    return "Osoba: " + first + " " + last + " " + address;
  }
  public static class NullPerson
  extends Person implements Null {
    private NullPerson() { super("None", "None", "None"); }
    public String toString() { return "NullPerson"; }
  }
  public static final Person NULL = new NullPerson();
} ///:~
```

Zasadniczo obiekt pusty byłby realizacją wzorca projektowego *Singleton*, więc tworzymy go tu jako egzemplarz statyczny i finalny. Całość zadziała, bo egzemplarze `Person` są niezmienne w czasie życia — wartości pól egzemplarza można ustawiać jedynie w wywołaniu konstruktora, ale nie da się ich potem zmieniać (podobnie jak nie da się zmodyfikować zawartości egzemplarza klasy `String`). Kiedy zechcesz zmienić obiekt `NullPerson`, możesz go jedynie zastąpić innym obiektem tej klasy. Zauważ, że dzięki `instanceof` masz też możliwość wykrycia podtypu `NullPerson` albo typu ogólniejszego `Null`; ale przy oparciu całości na wzorcu `Singleton` można też użyć metody `equals()` albo nawet operatora == w porównaniach z `Person.NULL`.

Wyobraź sobie, że znów mamy czasy gorączki internetowej i otrzymałeś właśnie od ochoczo inwestujących w e-biznes bankierów spory zastrzyk finansowy na rozwój własnego Niesamowitego Pomysłu. Jesteś gotów do uruchomienia spółki, czekasz tylko na uzupełnienia kadrowe; póki co możesz w miejsce każdego etatu (`Position`) wstawić pusty obiekt dla przyszłego pracownika (`Person`):

```
//: typeinfo/Position.java

class Position {
  private String title;
  private Person person;
  public Position(String jobTitle, Person employee) {
    title = jobTitle;
    person = employee;
    if(person == null)
      person = Person.NULL;
  }
  public Position(String jobTitle) {
    title = jobTitle;
    person = Person.NULL;
```

```
  }
  public String getTitle() { return title; }
  public void setTitle(String newTitle) {
    title = newTitle;
  }
  public Person getPerson() { return person; }
  public void setPerson(Person newPerson) {
    person = newPerson;
    if(person == null)
      person = Person.NULL;
  }
  public String toString() {
    return "Etat: " + title + " " + person;
  }
} ///:~
```

Nie musimy tworzyć osobnego obiektu pustego dla klasy `Position`, bo obecność `Person.NULL` implikuje nieobecność `Position` (być może później okaże się, że trzeba mimo wszystko uzupełnić projekt o pusty obiekt dla `Position`, ale reguła wstrzemięźliwości[5] (ang. *YAGNI — You Aren't Going to Need It*) mówi, żeby stworzyć „jedynie to, co niezbędne", a z rozbudową czekać do czasu, kiedy okaże się konieczna).

Klasa `Staff` (kadra) może teraz przy przydzielaniu etatów sprawdzać obecność obiektów pustych:

```
//: typeinfo/Staff.java
import java.util.*;

public class Staff extends ArrayList<Position> {
  public void add(String title, Person person) {
    add(new Position(title, person));
  }
  public void add(String... titles) {
    for(String title : titles)
      add(new Position(title));
  }
  public Staff(String... titles) { add(titles); }
  public boolean positionAvailable(String title) {
    for(Position position : this)
      if(position.getTitle().equals(title) &&
         position.getPerson() == Person.NULL)
        return true;
    return false;
  }
  public void fillPosition(String title, Person hire) {
    for(Position position : this)
      if(position.getTitle().equals(title) &&
         position.getPerson() == Person.NULL) {
        position.setPerson(hire);
        return;
      }
    throw new RuntimeException(
      "Brak etatu " + title);
```

[5] Doktryna programowania ekstremalnego (XP): „ogranicz się do możliwie najprostszego działającego rozwiązania".

```
  }
  public static void main(String[] args) {
    Staff staff = new Staff("Prezes", "Dyrektor zarządzający",
      "Kierownik marketingu", "Kierownik produktu",
      "Szef projektu", "Programista",
      "Programista", "Programista",
      "Programista", "Tester",
      "Opiekun dokumentacji");
    staff.fillPosition("Prezes",
      new Person("Ja", "Pierwszy", "Szczyt szczytów"));
    staff.fillPosition("Szef projektu",
      new Person("Janina", "Staranna", "Przedmieścia"));
    if(staff.positionAvailable("Programista"))
      staff.fillPosition("Programista",
        new Person("Robert", "Koder", "Piwniczna"));
    System.out.println(staff);
  }
} /* Output:
[Etat: Prezes Osoba: Ja Pierwszy Szczyt szczytów, Etat: Dyrektor zarządzający NullPerson, Etat:
Kierownik marketingu NullPerson, Etat: Kierownik produktu NullPerson, Etat: Szef projektu Osoba:
Janina Staranna Przedmieścia, Etat: Programista Osoba: Robert Koder Piwniczna, Etat: Programista
NullPerson, Etat: Programista NullPerson, Etat: Programista NullPerson, Etat: Tester NullPerson, Etat:
Opiekun dokumentacji NullPerson]
*///:~
```

Zauważ, że i tak tu i ówdzie trzeba testować obecność obiektów pustych, co nie różni się znów tak bardzo od wykrywania referencji pustych, ale w innych miejscach (tutaj zwłaszcza w konwersji `toString()`) dodatkowe testy są zbędne; można po prostu założyć, że wszystkie obiekty są poprawne, nawet jeśli niektóre nie reprezentują faktycznych egzemplarzy.

Przy pracy z interfejsami (zajmujących miejsce klas konkretnych) można automatycznie tworzyć obiekty puste za pośrednictwem dynamicznego proxy. Załóżmy, że dysponujemy interfejsem `Robot`, który definiuje nazwę, model i listę operacji `List<Operation>` opisujących zdolności robota. Klasa `Operation` zawiera opis operacji i właściwe polecenie (jak we wzorcu *Command*):

```
//: typeinfo/Operation.java

public interface Operation {
  String description();
  void command();
} ///:~
```

Do listy usług robota można się odwołać za pośrednictwem metody `operations()`:

```
//: typeinfo/Robot.java
import java.util.*;
import net.mindview.util.*;

public interface Robot {
  String name();
  String model();
  List<Operation> operations();
  class Test {
    public static void test(Robot r) {
      if(r instanceof Null)
        System.out.println("[Null Robot]");
```

```
      System.out.println("Nazwa robota: " + r.name());
      System.out.println("Model robota: " + r.model());
      for(Operation operation : r.operations()) {
        System.out.println(operation.description());
        operation.command();
      }
    }
  }
} ///:~
```

Tu również do testowania posłużyła zagnieżdżona klasa.

Teraz możemy utworzyć egzemplarz robota odśnieżającego:

```
//: typeinfo/SnowRemovalRobot.java
import java.util.*;

public class SnowRemovalRobot implements Robot {
  private String name;
  public SnowRemovalRobot(String name) {this.name = name;}
  public String name() { return name; }
  public String model() { return "ŚniegoBot Seria 11"; }
  public List<Operation> operations() {
    return Arrays.asList(
      new Operation() {
        public String description() {
          return name + " szufluje, odśnieża";
        }
        public void command() {
          System.out.println(name + " - szuflowanie");
        }
      },
      new Operation() {
        public String description() {
          return name + " kruszy lód";
        }
        public void command() {
          System.out.println(name + " - kruszenie");
        }
      },
      new Operation() {
        public String description() {
          return name + " czyści dachy";
        }
        public void command() {
          System.out.println(name + " - czyszczenie");
        }
      }
    );
  }
  public static void main(String[] args) {
    Robot.Test.test(new SnowRemovalRobot("Slusher"));
  }
} /* Output:
Nazwa robota: Slusher
Model robota: ŚniegoBot Seria 11
Slusher szufluje, odśnieża
Slusher - szuflowanie
```

Slusher kruszy lód
Slusher - kruszenie
Slusher czyści dachy
Slusher - czyszczenie
///:~

Zakładamy istnienie potencjalnie wielu podtypów klasy Robot i chcielibyśmy, aby każdy obiekt pusty robił coś zależnie od typu robota — w takim przypadku chodzi o pozyskanie informacji o dokładnym typie robota, w imieniu którego występuje obiekt pusty. Informacje te zostaną przechwycone w dynamicznym proxy:

```
//: typeinfo/NullRobot.java
// Użycie dynamicznego proxy do utworzenia obiektu pustego.
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;

class NullRobotProxyHandler implements InvocationHandler {
  private String nullName;
  private Robot proxied = new NRobot();
  NullRobotProxyHandler(Class<? extends Robot> type) {
    nullName = type.getSimpleName() + " NullRobot";
  }
  private class NRobot implements Null, Robot {
    public String name() { return nullName; }
    public String model() { return nullName; }
    public List<Operation> operations() {
      return Collections.emptyList();
    }
  }
  public Object
  invoke(Object proxy, Method method, Object[] args)
  throws Throwable {
    return method.invoke(proxied, args);
  }
}

public class NullRobot {
  public static Robot
  newNullRobot(Class<? extends Robot> type) {
    return (Robot)Proxy.newProxyInstance(
      NullRobot.class.getClassLoader(),
      new Class[]{ Null.class, Robot.class },
      new NullRobotProxyHandler(type));
  }
  public static void main(String[] args) {
    Robot[] bots = {
      new SnowRemovalRobot("SnowBee"),
      newNullRobot(SnowRemovalRobot.class)
    };
    for(Robot bot : bots)
      Robot.Test.test(bot);
  }
} /* Output:
Nazwa robota: SnowBee
Model robota: SniegoBot Seria 11
SnowBee szufluje, odśnieża
SnowBee - szuflowanie
```

```
SnowBee kruszy lód
SnowBee - kruszenie
SnowBee czyści dachy
SnowBee - czyszczenie
[Null Robot]
Nazwa robota: SnowRemovalRobot NullRobot
Model robota: SnowRemovalRobot NullRobot
*///:~
```

Kiedy trzeba utworzyć pusty obiekt klasy Robot, należy po prostu wywołać metodę `newNullRobot()`, przekazując w wywołaniu pożądany typ robota. Proxy wypełnia wymagania interfejsów `Robot` oraz `Null` i udostępnia nazwę typu, w którego obsłudze pośredniczy.

Imitacje i zalążki

Logicznymi wariantami wzorca projektowego obiektu pustego są imitacje obiektów (*Mock Object*) i zalążki (*Stub*); zadaniem obu jest czasowe zastępowanie „prawdziwego" obiektu, który zajmie ich miejsce w ostatecznym programie. Oba udają jednak „żywe" obiekty, zdolne do udostępniania rzeczywistych informacji — są więc czymś więcej niż zwykłymi atrapami zastępującymi referencje puste.

Różnica pomiędzy imitacją a zalążkiem to różnica ilościowa. Imitacje to zwykle obiekty proste i samotestujące i najczęściej tworzone właśnie do obsługi różnych aspektów testowania. Z kolei zalążki ograniczają się do zwracania szczątkowych danych, są zwykle rozbudowane i najczęściej wykorzystywane pomiędzy testami. Zalążki mogą być też konfigurowane pod kątem zmian zachowania w zależności od sposobu wywołania. Jak widać, zalążek to złożony obiekt wielozadaniowy; imitacje to z kolei niewielkie, specjalizowane, ale za to proste obiekty.

Ćwiczenie 24. Uzupełnij program *RegisteredFactories.java* o obiekty puste (4).

Interfejsy a RTTI

Ważnym zadaniem słowa kluczowego `interface` jest umożliwienie izolowania komponentów i rozluźniania zależności. Jeśli piszesz kod odnoszący się do interfejsu, zyskujesz ową izolację i rozluźnienie, ale nie znaczy to, że nie możesz tego uniknąć — wystarczy, że skorzystasz z informacji o typach. Interfejsy nie są więc absolutnymi gwarantami luźnych zależności. Oto przykład — zaczniemy od interfejsu:

```
//: typeinfo/interfacea/A.java
package typeinfo.interfacea;

public interface A {
  void f();
} ///:~
```

Dalej mamy implementację interfejsu; przy okazji zobaczymy, jak można prześlizgnąć się przez izolację interfejsu i dobrać do faktycznego typu implementacji:

```
//: typeinfo/InterfaceViolation.java
// Prześlizgiwanie się przez interfejs.
import typeinfo.interfacea.*;
```

```
class B implements A {
  public void f() {}
  public void g() {}
}

public class InterfaceViolation {
  public static void main(String[] args) {
    A a = new B();
    a.f();
    // a.g(); // Błąd kompilacji
    System.out.println(a.getClass().getName());
    if(a instanceof B) {
      B b = (B)a;
      b.g();
    }
  }
} /* Output:
B
*///:~
```

Za pomocą RTTI odkrywamy, że obiekt a to implementacja interfejsu A w postaci egzemplarza klasy B. Wykonując proste rzutowanie na B, możemy wywołać na rzecz wynikowego obiektu metody, które nie są dostępne w interfejsie A.

To całkowicie legalne i akceptowalne, niekiedy jednak chcemy uniknąć takiego sprytu programistów-klientów, bo zaprezentowana technika daje im możliwość wprowadzenia nazbyt silnego powiązania z naszym kodem. Sądzimy, że chroni nas przed tym słowo `interface`, tymczasem to nieprawda, a ujawnienie faktu, że do implementacji A służy B i przeniknięcie tej wiedzy do publicznej wiadomości to jedynie kwestia czasu[6].

Rozwiązaniem może być poinformowanie programistów, że jeśli zdecydują się na ominięcie blokady izolacyjnej i skorzystają z klasy, a nie zalecanego interfejsu, będą musieli radzić sobie z nią sami. To często najbardziej sensowne wyjście, ale jeśli nie wystarczy, trzeba wdrożyć silniejsze zabezpieczenia.

Najprościej wtedy nadać implementacji interfejsu dostęp pakietowy, tak aby była niewidoczna spoza pakietu:

```
//: typeinfo/packageaccess/HiddenC.java
package typeinfo.packageaccess;
import typeinfo.interfacea.*;
import static net.mindview.util.Print.*;

class C implements A {
  public void f() { print("publiczna metoda C.f()"); }
  public void g() { print("publiczna metoda C.g()"); }
  void u() { print("pakietowa metoda C.u()"); }
```

[6] Najgłośniejszym przypadkiem z tej dziedziny jest niewątpliwie system operacyjny Windows, w którym oficjalnie publikowanemu interfejsowi API, z którego programiści powinni korzystać, towarzyszyły nieoficjalne, ale widoczne funkcje, które można było odkryć i wywoływać. Programiści zaczęli powszechnie wykorzystywać owe ukryte funkcje API, co zmusiło firmę Microsoft do traktowania ich w ramach konserwacji kodu tak jak składników oficjalnego interfejsu. Dla firmy było to niewątpliwie bardzo kosztowne.

```
  protected void v() { print("chroniona metoda C.v()"); }
  private void w() { print("prywatna metoda C.w()"); }
}

public class HiddenC {
  public static A makeA() { return new C(); }
} ///:~
```

Jedyna publiczna część tego pakietu, w postaci klasy `HiddenC`, wytwarza implementację interfejsu `A`. Co ciekawe, mimo że wywołanie `makeA()` zwraca obiekt klasy `C`, to poza macierzystym pakietem nie można go użyć do wywołania metod spoza interfejsu `A`.

Jeśli teraz ktoś spróbuje rzutować otrzymaną implementację w dół, na typ `C`, nie uda mu się to, bo typ `C` poza pakietem jest niedostępny:

```
//: typeinfo/HiddenImplementation.java
// Wymijanie blokady w postaci dostępu pakietowego.
import typeinfo.interfacea.*;
import typeinfo.packageaccess.*;
import java.lang.reflect.*;

public class HiddenImplementation {
  public static void main(String[] args) throws Exception {
    A a = HiddenC.makeA();
    a.f();
    System.out.println(a.getClass().getName());
    // Błąd kompilacji: cannot find symbol 'C':
    /* if(a instanceof C) {
      C c = (C)a;
      c.g();
    } */
    // Oho! Mechanizm refleksji wciąż pozwala na wywołanie g():
    callHiddenMethod(a, "g");
    // A nawet jeszcze mniej dostępnych metod!
    callHiddenMethod(a, "u");
    callHiddenMethod(a, "v");
    callHiddenMethod(a, "w");
  }
  static void callHiddenMethod(Object a, String methodName)
  throws Exception {
    Method g = a.getClass().getDeclaredMethod(methodName);
    g.setAccessible(true);
    g.invoke(a);
  }
} /* Output:
publiczna metoda C.f()
typeinfo.packageaccess.C
publiczna metoda C.g()
pakietowa metoda C.u()
chroniona metoda C.v()
prywatna metoda C.w()
*///:~
```

Jak widać, wciąż — tym razem za pośrednictwem refleksji — można dobrać się do metod, i to *wszystkich*, nawet prywatnych! Wystarczy znać nazwę metody, aby wywołać `setAccesible(true)` na rzecz obiektu `Method` i tym samym uzdatnić metodę do wywołania — widać to w `callHiddenMethod()`.

Zdawałoby się, że można temu zapobiec, rozprowadzając wyłącznie kod skompilowany, ale to również nie jest żadnym rozwiązaniem. Wystarczy przecież skorzystać z programu javap dekompilatora wchodzącego w skład zestawu JDK. Wystarczy takie polecenie:

```
javap -private C
```

Opcja `-private` to żądanie wypisania wszystkich składowych, z prywatnymi włącznie. Oto wynik wydania takiego polecenia:

```
Compiled from "HiddenC.java"
class typeinfo.packageaccess.C extends
java.lang.Object implements typeinfo.interfacea.A {
    typeinfo.packageaccess.C();
    public void f();
    public void g();
    void u();
    protected void v();
    private void w();
}
```

Jak widać, każdy może dobrać się do nazw i sygnatur najbardziej nawet prywatnych metod, aby je potem wywołać.

A co, jeśli zaimplementujemy interfejs jako prywatną klasę wewnętrzną? Wyglądałoby to tak:

```
// : typeinfo/InnerImplementation.java
// Prywatna klasa wewnętrzna też nie ukryje się przed refleksją.
import typeinfo.interfacea.*;
import static net.mindview.util.Print.*;

class InnerA {
  private static class C implements A {
    public void f() { print("publiczna metoda C.f()"); }
    public void g() { print("publiczna metoda C.g()"); }
    void u() { print("pakietowa metoda C.u()"); }
    protected void v() { print("chroniona metoda C.v()"); }
    private void w() { print("prywatna metoda C.w()"); }
  }
  public static A makeA() { return new C(); }
}

public class InnerImplementation {
  public static void main(String[] args) throws Exception {
    A a = InnerA.makeA();
    a.f();
    System.out.println(a.getClass().getName());
    // Refleksja i tak udostępni prywatną klasę:
    HiddenImplementation.callHiddenMethod(a, "g");
    HiddenImplementation.callHiddenMethod(a, "u");
    HiddenImplementation.callHiddenMethod(a, "v");
    HiddenImplementation.callHiddenMethod(a, "w");
  }
} /* Output:
publiczna metoda C.f()
InnerA$C
publiczna metoda C.g()
pakietowa metoda C.u()
```

```
chroniona metoda C.v()
prywatna metoda C.w()
*///:~
```

I to nie ukryło niczego przed wszędobylską refleksją. A może klasa anonimowa?

```
//: typeinfo/AnonymousImplementation.java
// Nawet klasa anonimowa nie zabezpiecza przed refleksją.
import typeinfo.interfacea.*;
import static net.mindview.util.Print.*;

class AnonymousA {
  public static A makeA() {
    return new A() {
      public void f() { print("publiczna metoda C.f()"); }
      public void g() { print("publiczna metoda C.g()"); }
      void u() { print("pakietowa metoda C.u()"); }
      protected void v() { print("chroniona metoda C.v()"); }
      private void w() { print("prywatna metoda C.w()"); }
    };
  }
}

public class AnonymousImplementation {
  public static void main(String[] args) throws Exception {
    A a = AnonymousA.makeA();
    a.f();
    System.out.println(a.getClass().getName());
    // Refleksja i tak udostępni klasę anonimową:
    HiddenImplementation.callHiddenMethod(a, "g");
    HiddenImplementation.callHiddenMethod(a, "u");
    HiddenImplementation.callHiddenMethod(a, "v");
    HiddenImplementation.callHiddenMethod(a, "w");
  }
} /* Output:
publiczna metoda C.f()
AnonymousA$1
publiczna metoda C.g()
pakietowa metoda C.u()
chroniona metoda C.v()
prywatna metoda C.w()
*///:~
```

Zdaje się, że wskutek refleksji nie ma sposobu, aby zapobiec sięganiu do metod o dostępie niepublicznym. Dotyczy to również pól klasy, nawet tych prywatnych:

```
//: typeinfo/ModifyingPrivateFields.java
import java.lang.reflect.*;

class WithPrivateFinalField {
  private int i = 1;
  private final String s = "Jestem zupełnie bezpieczny";
  private String s2 = "A ja jestem bezpieczny?";
  public String toString() {
    return "i = " + i + ", " + s + ", " + s2;
  }
}
```

```
public class ModifyingPrivateFields {
  public static void main(String[] args) throws Exception {
    WithPrivateFinalField pf = new WithPrivateFinalField();
    System.out.println(pf);
    Field f = pf.getClass().getDeclaredField("i");
    f.setAccessible(true);
    System.out.println("f.getInt(pf): " + f.getInt(pf));
    f.setInt(pf, 47);
    System.out.println(pf);
    f = pf.getClass().getDeclaredField("s");
    f.setAccessible(true);
    System.out.println("f.get(pf): " + f.get(pf));
    f.set(pf, "Nie, nie jesteś!");
    System.out.println(pf);
    f = pf.getClass().getDeclaredField("s2");
    f.setAccessible(true);
    System.out.println("f.get(pf): " + f.get(pf));
    f.set(pf, "Nie, nie jesteś!");
    System.out.println(pf);
  }
} /* Output:
i = 1, Jestem zupełnie bezpieczny, A ja jestem bezpieczny?
f.getInt(pf): 1
i = 47, Jestem zupełnie bezpieczny, A ja jestem bezpieczny?
f.get(pf): Jestem zupełnie bezpieczny
i = 47, Jestem zupełnie bezpieczny, A ja jestem bezpieczny?
f.get(pf): A ja jestem bezpieczny?
i = 47, Jestem zupełnie bezpieczny, Nie, nie jesteś!
*///:~
```

Bezpieczne są w zasadzie jedynie pola finalne — w tym sensie, że nie uda się ich zmodyfikować. System wykonawczy zaakceptuje co prawda każdą próbę modyfikacji bez sprzeciwu, ale i tak do niej nie dojdzie.

Ogólnie rzecz biorąc, zaprezentowane naruszenia dostępu, choć pozornie groźne, nie są takie straszne. Jeśli ktoś ucieka się do takich technik w celu wywołania metod, które projektant przewidział jako prywatne, albo przeznaczone do użytku wewnątrz pakietu, to jeśli postanowisz w przyszłości, zmienione zostaną niektóre aspekty zachowania tych metod, będzie mógł mieć pretensje wyłącznie do siebie. Z drugiej strony fakt, że do każdej klasy można się dobrać tylnymi drzwiami, pozwala na rozwiązywanie niektórych problemów, które inaczej byłyby trudne albo wprost niemożliwe do rozwiązania — dlatego refleksję należy uznać za raczej pożądaną.

Ćwiczenie 25. Utwórz klasę zawierającą metody prywatne, chronione i dostępne w obrębie pakietu. Napisz kod, który wywoła wszystkie te metody spoza pakietu klasy (2).

Podsumowanie

RTTI pozwala na uzyskanie informacji o typie na podstawie anonimowego odwołania do klasy bazowej. Tak więc możliwe są nadużycia, szczególnie w przypadku nowicjuszy, gdyż użycie tego mechanizmu może się wydawać sensowniejsze od wywołania metody polimorficznej. Dla wielu osób posiadających doświadczenie w projektowaniu w językach

proceduralnych trudne jest zorganizowanie programu inaczej niż jako zestawu instrukcji `switch`. Mogą to osiągnąć dzięki RTTI i tym samym zatracić istotę polimorfizmu w tworzeniu i utrzymywaniu kodu. Intencją programowania obiektowego jest stosowanie wywołań metod polimorficznych w całym kodzie i stosowanie RTTI tylko wtedy, gdy jest to konieczne.

Jednak stosowanie wywołań polimorficznych metod zgodnie z zamiarami wymaga kontroli nad definicją klasy bazowej, ponieważ w pewnych sytuacjach przy rozszerzeniu programu można odkryć, że klasa bazowa nie zawiera potrzebnej metody. Jeżeli klasa bazowa pochodzi z zewnętrznej biblioteki albo jest kontrolowana przez kogoś innego, to jednym z rozwiązań jest RTTI: można mianowicie stworzyć nowy typ pochodny i dołożyć własną metodę. W innym miejscu w kodzie można natomiast wykrywać ten nowy typ i wywoływać dodaną metodę. Nie burzy to polimorfizmu i rozszerzalności programu, ponieważ dodanie nowego typu nie wymaga zmian we wszystkich wyrażeniach `switch`. Jednak jeżeli dołożymy nowy kod w głównej części programu, która wymaga nowych funkcji, to trzeba wykorzystać RTTI, aby wykryć nasz typ.

Dołożenie funkcji do klasy bazowej może oznaczać, że dla korzyści jednej konkretnej klasy wszystkie inne, dziedziczące z tej bazowej, wymagają implementacji nowej metody. Czyni to interfejs mniej zrozumiałym i drażni tych, którzy muszą przesłaniać metody abstrakcyjne, dziedzicząc z klasy bazowej. Przykładowo rozważmy hierarchię klas reprezentującą instrumenty muzyczne. Załóżmy, że chcemy przeczyścić ustniki określonych instrumentów orkiestrowych. Jedną z opcji jest zamieszczenie metody `clearSpitValve()` w klasie bazowej `Instrument`, ale byłoby to rozwiązanie mylące, ponieważ oznaczałoby, że instrumenty klasy `Percussion`, `Stringed` i `Electronic` także posiadają jakieś ustniki. W tym przypadku RTTI zapewnia znacznie bardziej sensowne rozwiązanie, ponieważ można zamieścić metodę w określonej klasie (w tym przypadku w klasie instrumentów dętych `Wind`), w której byłaby już właściwa. Jeszcze właściwszym rozwiązaniem jest zamieszczenie w klasie bazowej metody przygotowującej instrument `prepareInstrument()`, choć podczas pierwszego podejścia do problemu można tego nie zauważyć i błędnie przyjmować konieczność wykorzystania RTTI.

Użycie RTTI czasami może rozwiązać problemy z wydajnością. Jeżeli kod poprawnie wykorzystuje polimorfizm, ale okaże się, że jeden z obiektów zareaguje na to uniwersalne wywołanie w bardzo niewydajny sposób, to można wykryć taki typ, stosując mechanizm identyfikacji i napisać kod specyficzny dla tego przypadku. Należy jednak być ostrożnym w przypadku zbyt wczesnego zajmowania się wydajnością — jest to kusząca pułapka. Lepiej *najpierw* napisać działający program, a potem zdecydować, czy działa on wystarczająco szybko, i tylko wtedy rozwiązywać kwestie wydajności — koniecznie z narzędziem do profilowania kodu (patrz suplement publikowany pod adresem *http://MindView/Books/BetterJava*).

Przekonaliśmy się przy okazji, że mechanizm refleksji otwiera całkiem nowe możliwości programistyczne, dopuszczając programowanie o znacznie bardziej dynamicznym charakterze. Są tacy, którym dynamiczna natura refleksji przeszkadza. Możliwość wykonywania operacji, które mogą być kontrolowane jedynie w czasie wykonania, a ich niepowodzenie sygnalizowane tylko wyjątkami, jest dla programistów przywykłych do komfortu statycznej kontroli typów czymś nienaturalnym. Niektórzy stwierdzają wprost, że wyjątek czasu wykonania jest jasnym sygnałem, że powodującego go kodu należałoby unikać.

Osobiście uważam, że takie poczucie bezpieczeństwa jest iluzyjne — zawsze będą przecież takie elementy, które mogą się pojawić jedynie w czasie wykonania i spowodować wtedy wyjątek nawet w programie niezawierającym ani jednego bloku `try` i ani jednej specyfikacji wyjątków. Uważam więc obecność modelu zgłaszania błędów czasu wykonania *za silne wsparcie* przy pisaniu kodu wykorzystującego refleksję. Oczywiście warto zabiegać o statyczną kontrolę kodu... ale tylko tam, gdzie to jest możliwe. Ja zaś uważam, że kod dynamiczny to jedna z większych zalet Javy wobec języków takich jak C++.

Ćwiczenie 26. Zaimplementuj metodę `clearSpitValve()` zgodnie z opisem w podsumowaniu (3).

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 15.
Typy ogólne

Zwykłe klasy i metody operują na konkretnych, określonych typach — czy to podstawowych, czy to klasach — ale przy pisaniu kodu przeznaczonego do operowania na wielu typach ten brak elastyczności metod i klas może być uciążliwy[1].

Uogólnienie rozwiązań w modelach obiektowych osiąga się poprzez polimorfizm. Polega to na pisaniu metody (na przykład), która przyjmuje argument w postaci obiektu klasy bazowej; do takiej metody przekazuje się potem obiekty klas pochodnych wyprowadzonych z owej klasy bazowej. Taka metoda jest nieco bardziej ogólna (w znaczeniu: „uniwersalna"), przez co nadaje się do stosowania w większej liczbie kontekstów. To samo dotyczy wnętrza klas — wszędzie tam, gdzie w jej definicji występuje operacja angażująca konkretny typ, można pokusić się o podstawienie w jego miejsce typu bazowego, zwiększając „zasięg" operacji. A skoro wszystkie klasy (poza finalnymi[2]) dają się rozszerzać przez dziedziczenie, elastyczność zyskujemy niejako automatycznie.

Niekiedy jednak ograniczenie uniwersalności operacji do pojedynczej hierarchii klas jest zbyt dotkliwe. Jeśli typ argumentu metody określany jest przez interfejs, a nie klasę, ograniczenie to rozluźnia się, bo w takim układzie metoda może przyjmować obiekty dowolnego typu implementującego interfejs. Programista-klient może więc uzdatniać swoje typy do użycia w danej metodzie przez implementowanie w nich wymaganych interfejsów. Interfejsy pozwalają więc na wykroczenie poza hierarchie klas.

Niekiedy jednak nawet konieczność implementowania interfejsu okazuje się mocnym ograniczeniem. Implementacja interfejsu oznacza bowiem konieczność obsługiwania tegoż konkretnego interfejsu. Tymczasem łatwo wyobrazić sobie kod jeszcze bardziej ogólny, który operowałby nie na konkretnej klasie czy interfejsie, ale na „jakimś nieokreślonym bliżej typie".

---

[1] Przy przygotowywaniu niniejszego rozdziału czerpałem garściami z *Java Generics FAQ* Angeliki Langer (zobacz *www.angelikalanger.com*) i innych jej publikacji (również tych powstałych z udziałem Klausa Krefta).

[2] I klasami z prywatnymi konstruktorami.

Tak przedstawia się koncepcja typów ogólnych, stanowiących jedną z poważniejszych zmian w Javie SE5 wobec wydań poprzednich. Uogólnienia są realizacją koncepcji *parametryzacji typów*, która zakłada możliwość tworzenia komponentów (zwykle chodzi o kontenery) nadających się do stosowania z wieloma rozmaitymi typami. Pojęcie „ogólności" ma tu znaczenie „odpowiedniości dla szerokiego grona klas". Pierwotnie uogólnienia w językach programowania służyły programiście w roli środków poszerzania znaczenia pisanych klas i metod, przez rozluźnienie ograniczeń narzucanych typom, z którymi owe klasy i metody miały pracować. Jak się przekonasz, implementacja uogólnień w Javie nie sięga aż tak daleko — można nawet poddawać w wątpliwość zasadność użycia tego terminu.

Jeśli nie miałeś wcześniej do czynienia z żadnym mechanizmem parametryzacji typu, uogólnienia Javy będą dla Ciebie zapewne wygodnym uzupełnieniem języka. Docenisz to, że przy tworzeniu egzemplarza typu sparametryzowanego dochodzi do automatycznego rzutowania przy zachowaniu kontroli poprawności typu w czasie kompilacji. A więc całkiem nieźle.

Ale kto zna mechanizm parametryzacji typów choćby w wydaniu języka C++, uogólnieniami dostępnymi w Javie może się cokolwiek rozczarować. Wykorzystywanie typu uogólnionego, utworzonego przez osobę trzecią, jest jeszcze w miarę wygodne, ale tworząc własne typy sparametryzowane, napotkasz szereg niespodzianek. Będę się więc starał wyjaśniać między innymi to, skąd wzięła się taka, a nie inna postać uogólnień w Javie.

Nie chcę przez to powiedzieć, że uogólnienia są w Javie bezużyteczne. W wielu przypadkach pozwalają na uproszczenie i zwiększenie przejrzystości, a nawet elegancji kodu. Ale kto zna język implementujący czystszą formę uogólnień, dostrzeże liczne ograniczenia Javy w tym aspekcie. W rozdziale przyjrzymy się zarówno zaletom, jak i ograniczeniom uogólnień Javy — wszak chodzi o to, abyś mógł ich potem efektywnie używać.

Porównanie z językiem C++

Projektanci Javy niewątpliwie czerpali inspirację z języka C++. Mimo to można z powodzeniem nauczać programowania w Javie bez powoływania się na liczne analogie z C++; tak właśnie staram się prowadzić omówienie, czyniąc wyjątki jedynie tam, gdzie takie porównanie może przyczynić się do lepszego ogarnięcia omawianej kwestii.

Uogólnienia bardziej niż co innego nadają się do takiego porównania. Po pierwsze, ogarnięcie niektórych aspektów *szablonów* (ang. *templates*) języka C++ (stanowiących wzór dla uogólnień, również w zakresie podstawowej składni) pomaga w zrozumieniu założeń i podstaw całej koncepcji oraz — co bardzo ważne — zrozumieniu ograniczeń, jakich należy spodziewać się w Javie, i ich przyczyn. W ostatecznym efekcie chciałbym wyposażyć Cię w wiedzę o położeniu owych granic, bo z mojego doświadczenia wynika, że świadomość i rozumienie sensu tych granic zwiększa możliwości programisty. Wiedząc, czego nie można w dany sposób osiągnąć, można od razu skupić się na rzeczach dostępnych, bez marnowania czasu na rozbijanie głową murów.

Po drugie zaś, społeczność programistów Javy przejawia (jako ogół) zasadnicze niezrozumienie co do szablonów C++, co potem negatywnie wpływa na wyobrażenie co do istoty i zadań typów ogólnych.

Z wymienionych przyczyn w rozdziale znajdzie się kilka przykładów szablonów w języku C++, ale dosłownie kilka i tylko w niezbędnym zakresie.

Proste uogólnienia

Jedną z głównym motywacji dla powołania do życia uogólnień była wizja tworzenia *klas kontenerów* (o których mówiliśmy w rozdziale „Kolekcje obiektów" i powiemy sobie więcej w rozdziale „Kontenery z bliska"). Kontener to obiekt przechowujący inne obiekty, poddawane manipulacjom w programie. Choć taka definicja kontenera obejmuje również zwyczajne tablice, kontenery przejawiają większą elastyczność od tablic, wyróżniają się też wieloma aspektami. Niemal każdy nietrywialny program wymaga przechowywania pewnej liczby obiektów wykorzystywanych w programie; dlatego też kontenery to jedne z częściej wykorzystywanych klas bibliotecznych.

Przyjrzyjmy się klasie zdolnej do przechowywania pojedynczego obiektu. Klasa taka mogłaby określać typ obiektu przechowywanego jawnie, jak tu:

```
//: generics/Holder1.java

class Automobile {}

public class Holder1 {
  private Automobile a;
  public Holder1(Automobile a) { this.a = a; }
  Automobile get() { return a; }
} ///:~
```

Ale takie narzędzie nie jest specjalnie uniwersalne, bo nie można w nim przechowywać niczego poza obiektami `Automobile`. Dla każdego typu przechowywanych obiektów musielibyśmy pisać osobne klasy kontenerów.

Przed pojawieniem się Javy SE5 moglibyśmy określić typ obiektu przechowywanego jako `Object`:

```
//: generics/Holder2.java

public class Holder2 {
  private Object a;
  public Holder2(Object a) { this.a = a; }
  public void set(Object a) { this.a = a; }
  public Object get() { return a; }
  public static void main(String[] args) {
    Holder2 h2 = new Holder2(new Automobile());
    Automobile a = (Automobile)h2.get();
    h2.set("Niekoniecznie Automobile");
    String s = (String)h2.get();
```

```
    h2.set(1); // Automatyczne pakowanie w obiekcie Integer
    Integer x = (Integer)h2.get();
  }
} ///:~
```

Kontener klasy `Holder2` może przechowywać dosłownie wszystko — w powyższym przykładzie występował w roli przechowalni trzech obiektów różnych typów.

Są takie sytuacje, w których chcielibyśmy zapewnić sobie możliwość wykorzystania kontenera do przechowywania obiektów wielu typów, ale w większości przypadków użycia będą to obiekty jednego typu. Jedną z przyczyn włączenia uogólnień do języka była chęć określania typu obiektów przechowywanych przez kontener, tak aby kompilator wspierał programistę w kontroli typów umieszczanych tam obiektów.

Zamiast podawać jako typ klasę `Object`, chcemy pozostawić niedookreślony typ elementów przechowywanych, tak aby można go było sprecyzować później, w miejscu użycia kontenera. W tym celu należy w definicji klasy, za jej nazwą w nawiasach kątowych umieścić parametr typowy, a w miejscu użycia klasy zastąpić go właściwym typem. W przypadku klasy naszego kontenera wyglądałoby to tak (parametrem typowym jest tu T):

```
//: generics/IIolder3.java

public class Holder3<T> {
  private T a;
  public Holder3(T a) { this.a = a; }
  public void set(T a) { this.a = a; }
  public T get() { return a; }
  public static void main(String[] args) {
    Holder3<Automobile> h3 =
      new Holder3<Automobile>(new Automobile());
    Automobile a = h3.get(); // Bez rzutowania
    // h3.set("Niekoniecznie Automobile"); // Błąd
    // h3.set(1); // Błąd
  }
} ///:~
```

Teraz przy tworzeniu egzemplarza klasy `Holder3` trzeba podać typ obiektów, które mają być w nim przechowywane, znów korzystając z nawiasów kątowych (zobacz metodę `main()`). Tak utworzony kontener może przechowywać jedynie obiekty podanego typu (albo jego podtypów, bo uogólnienia nie znoszą zasady zastępowania typów bazowych typami pochodnymi). A „wyjęcie" obiektu z kontenera nie musi już być połączone z rzutowaniem.

Tak przedstawia się w wielkim skrócie koncepcja uogólnień w języku Java: w miejscu użycia klasy doprecyzowuje się jej typ, a resztą zajmuje się kompilator.

Zasadniczo typy ogólne można traktować tak jak wszystkie inne typy — różnią się od nich tylko obecnością parametru typowego. Przekonasz się wkrótce, że uogólnienia można wykorzystywać jedynie przez podanie nazwy z listą argumentów typowych.

Ćwiczenie 1. Użyj kontenera `Holder3` w połączeniu z biblioteką `typeinfo.pets` do pokazania, że egzemplarz `Holder3` specjalizowany dla typu bazowego biblioteki nadaje się do przechowywania również obiektów typów pochodnych (1).

Ćwiczenie 2. Utwórz klasę kontenera przechowującego trzy obiekty tego samego typu oraz metody do składowania i wybierania obiektów przechowywanych wraz z konstruktorem inicjalizującym wszystkie trzy obiekty (1).

Biblioteka krotek

Niejednokrotnie z metody chciałoby się zwrócić więcej niż jedną wartość (obiekt). Instrukcja return pozwala jednak na określenie tylko jednej wartości zwracanej, więc aby zwrócić większą ich liczbę, trzeba stworzyć na potrzeby tej instrukcji obiekt zawierający ileś wartości. Oczywiście zadanie to można realizować za każdym razem od nowa, pisząc wedle potrzeb specjalne klasy; uogólnienia pozwalają jednak na rozwiązanie problemu raz na zawsze i to przy każdorazowym zachowaniu bezpieczeństwa wynikającego ze statycznej kontroli typów.

Chodzi nam o tak zwaną *krotkę* (ang. *tuple*), czyli grupę obiektów ujętych w innym obiekcie. Odbiorca takiego obiektu może odczytać z niego zawarte w nim elementy, ale nie może umieszczać nowych; koncepcję tę opisuje wzorzec projektowy *Data Transfer Object* („obiekt transferu danych") tudzież *Messenger* („posłaniec").

Krotki mogą mieć zwykle dowolne rozmiary, a każdy obiekt składowy może być innego typu. Chcemy jednak mieć możliwość określenia tych typów, tak aby odbiorca krotki mógł przy odczycie wartości polegać na ich typach. Problem różnicowania ilości składowych krotki rozwiązujemy przez osobne implementacje krotek. Oto krotka przeznaczona dla par obiektów:

```
//: net/mindview/util/TwoTuple.java
package net.mindview.util;

public class TwoTuple<A,B> {
  public final A first;
  public final B second;
  public TwoTuple(A a, B b) { first = a; second = b; }
  public String toString() {
    return "(" + first + ", " + second + ")";
  }
} ///:~
```

Konstruktor krotki pobiera obiekty składowe, a metoda `toString()` służy do wygodnego wypisywania zawartości krotki. Zauważ, że krotka niejawnie zachowuje kolejność elementów.

Po pierwszym czytaniu możesz dojść do wniosku, że powyższy kod narusza ogólne zasady bezpieczeństwa programowania w Javie. Czy składowe `first` i `second` nie powinny być prywatne, z możliwością odwołań za pośrednictwem metod `getFirst()` i `getSecond()`? Rozważ jednak wpływ takiej zmiany na bezpieczeństwo danych w krotce: użytkownicy i tak mogliby odczytywać wartości obiektów składowych i robić z nimi, co im przyjdzie do głowy, ale wciąż nie mogliby niczego przypisać do `first` ani `second`. Słowo `final` w deklaracjach pól krotki daje nam podobny poziom bezpieczeństwa, a do tego całość jest prostsza i krótsza.

Można też zauważyć, że w niektórych sytuacjach możliwość przypisywania wartości do pól krotki byłaby pożądana, ale bezpieczniej zostawić krotkę w obecnej postaci i po prostu zmusić użytkownika do podmieniania wartości składowych przez tworzenie nowych krotek `TwoTuple`.

Krotki o większej liczbie elementów można tworzyć w wyniku dziedziczenia. Dodawanie kolejnych parametrów typowych jest bardzo proste:

```
//: net/mindview/util/ThreeTuple.java
package net.mindview.util;

public class ThreeTuple<A,B,C> extends TwoTuple<A,B> {
  public final C third;
  public ThreeTuple(A a, B b, C c) {
    super(a, b);
    third = c;
  }
  public String toString() {
    return "(" + first + ", " + second + ", " + third +")";
  }
} ///:~

//: net/mindview/util/FourTuple.java
package net.mindview.util;

public class FourTuple<A,B,C,D> extends ThreeTuple<A,B,C> {
  public final D fourth;
  public FourTuple(A a, B b, C c, D d) {
    super(a, b, c);
    fourth = d;
  }
  public String toString() {
    return "(" + first + ", " + second + ", " +
      third + ", " + fourth + ")";
  }
} ///:~

//: net/mindview/util/FiveTuple.java
package net.mindview.util;

public class FiveTuple<A,B,C,D,E>
extends FourTuple<A,B,C,D> {
  public final E fifth;
  public FiveTuple(A a, B b, C c, D d, E e) {
    super(a, b, c, d);
    fifth = e;
  }
  public String toString() {
    return "(" + first + ", " + second + ", " +
      third + ", " + fourth + ", " + fifth + ")";
  }
} ///:~
```

Aby użyć krotki, wystarczy jako wartość zwracaną metody zadeklarować krotkę o odpowiedniej liczbie składowych, a następnie w metodzie utworzyć egzemplarz krotki i zwrócić ją do wywołującego instrukcją `return`:

```
//: generics/TupleTest.java
import net.mindview.util.*;

class Amphibian {}
class Vehicle {}

public class TupleTest {
  static TwoTuple<String,Integer> f() {
    // Automatyczne pakowanie w obiekty konwertuje int na Integer:
    return new TwoTuple<String,Integer>("hej", 47);
  }
  static ThreeTuple<Amphibian,String,Integer> g() {
    return new ThreeTuple<Amphibian, String, Integer>(
      new Amphibian(), "hej", 47);
  }
  static
  FourTuple<Vehicle,Amphibian,String,Integer> h() {
    return
      new FourTuple<Vehicle,Amphibian,String,Integer>(
        new Vehicle(), new Amphibian(), "hej", 47);
  }
  static
  FiveTuple<Vehicle,Amphibian,String,Integer,Double> k() {
    return new
      FiveTuple<Vehicle,Amphibian,String,Integer,Double>(
        new Vehicle(), new Amphibian(), "hej", 47, 11.1);
  }
  public static void main(String[] args) {
    TwoTuple<String,Integer> ttsi = f();
    System.out.println(ttsi);
    // ttsi.first = "tutaj"; // Błąd kompilacji: pole finalne
    System.out.println(g());
    System.out.println(h());
    System.out.println(k());
  }
} /* Output: (80% match)
(hej, 47)
(Amphibian@1f6a7b9, hej, 47)
(Vehicle@35ce36, Amphibian@757aef, hej, 47)
(Vehicle@9cab16, Amphibian@1a46e30, hej, 47, 11.1)
*///:~
```

Dzięki uogólnieniom można łatwo tworzyć krotki dowolnych typów, w celu zwracania za ich pośrednictwem grup dowolnych obiektów.

Widać tu, że modyfikator `final` opatrujący pola publiczne zapobiega przypisywaniu wartości do tych pól już po konstrukcji egzemplarza — dowodzi tego niepowodzenie instrukcji `ttsi.first = "tutaj"`.

Wyrażenia z operatorem `new` są tu nieco rozwlekłe. Niebawem poznasz sposób ich uproszczenia za pośrednictwem *metod uogólnionych*.

Ćwiczenie 3. Utwórz i przetestuj typ uogólniony `SixTuple` (krotkę sześcioelementową) (1).

Ćwiczenie 4. „Uogólnij" klasę z pliku *innerclasses/Sequence.java* (3).

Klasa stosu

Weźmy się za coś bardziej skomplikowanego: implementację tradycyjnego stosu elementów. W rozdziale „Kolekcje obiektów" mieliśmy okazję analizować implementację stosu na bazie kontenera `LinkedList` w klasie `net.mindview.util.Stack`. W tamtym przykładzie przekonałeś się, że wszystkie metody potrzebne nam do utworzenia interfejsu stosu udostępnia klasa `LinkedList`. Sam stos został skonstruowany ze złożenia klasy uogólnionej (`Stack<T>`) z inną klasą uogólnioną (`LinkedList<T>`). Przekonaliśmy się wtedy, że (poza nielicznymi wyjątkami, o których później) typ uogólniony to po prostu kolejny, zwyczajny typ.

Tym razem zaimplementujemy stos na bazie własnej, wewnętrznej implementacji listy.

```
//: generics/LinkedStack.java
// Stos implementowany na bazie własnej, wewnętrznej listy.

public class LinkedStack<T> {
  private static class Node<U> {
    U item;
    Node<U> next;
    Node() { item = null; next = null; }
    Node(U item, Node<U> next) {
      this.item = item;
      this.next = next;
    }
    boolean end() { return item == null && next == null; }
  }
  private Node<T> top = new Node<T>(); // Znacznik końca
  public void push(T item) {
    top = new Node<T>(item, top);
  }
  public T pop() {
    T result = top.item;
    if(!top.end())
      top = top.next;
    return result;
  }
  public static void main(String[] args) {
    LinkedStack<String> lss = new LinkedStack<String>();
    for(String s : "Ustawić fazery na ogłuszanie!".split(" "))
      lss.push(s);
    String s;
    while((s = lss.pop()) != null)
      System.out.println(s);
  }
} /* Output:
ogłuszanie!
na
fazery
Ustawić
*///:~
```

Wewnętrzna klasa `Node` (węzeł listy) również jest klasą uogólnioną i posiada własny parametr typowy.

W przykładzie uciekliśmy się do zastosowania znacznika końca (ang. *end sentinel*), czyli wyróżnionej wartości sygnalizującej brak elementów na stosie. Wartość ta jest tworzona przy okazji konstrukcji egzemplarza `LinkedStack`; z kolei każde wywołanie metody `push()`, odkładające element na stos, powoduje utworzenie nowego węzła `Node<T>` i dowiązanie go do poprzedniego. Metoda `pop()` zawsze zwraca element szczytowy stosu `top.item`, po czym porzuca bieżący węzeł `Node<T>` i przechodzi do następnego elementu, który od tego momentu jest elementem szczytowym — chyba że poprzednim elementem był znacznik końca. W ten sposób powtarzanie wywołania `pop()` po zdjęciu ze stosu wszystkich elementów będzie powodowało zwracanie wartości `null` sygnalizującej stan stosu.

Ćwiczenie 5. Usuń parametr typowy z klasy `Node` i zmodyfikuj resztę kodu programu *LinkedStack.java* tak, aby pokazać, że klasa wewnętrzna ma dostęp do parametrów typowych uogólnienia klasy otaczającej (2).

RandomList

W ramach następnego przykładu z kontenerem załóżmy, że potrzebujemy listy, ale takiej, która w reakcji na wywołanie metody `select()` losowo wybiera i zwraca któryś z elementów listy. Implementacja ma być narzędziem uniwersalnym, więc trzeba zastosować uogólnienia:

```
//: generics/RandomList.java
import java.util.*;

public class RandomList<T> {
  private ArrayList<T> storage = new ArrayList<T>();
  private Random rand = new Random(47);
  public void add(T item) { storage.add(item); }
  public T select() {
    return storage.get(rand.nextInt(storage.size()));
  }
  public static void main(String[] args) {
    RandomList<String> rs = new RandomList<String>();
    for(String s: ("Pchnąć w tę łódź jeża " +
        "lub ośm skrzyń fig").split(" "))
      rs.add(s);
    for(int i = 0; i < 11; i++)
      System.out.print(rs.select() + " ");
  }
} /* Output:
tę fig jeża tę jeża tę w lub Pchnąć jeża łódź
*///:~
```

Ćwiczenie 6. Użyj klasy `RandomList` z jeszcze dwoma typami (poza tym wykorzystanym w metodzie `main()`) (1).

Uogólnianie interfejsów

Typy ogólne współgrają również z interfejsami. Weźmy jako przykład *generator*, czyli klasę wytwarzającą obiekty. W istocie będzie to realizacja wzorca projektowego *Factory Method* („metoda wytwórcza"), tyle że żądanie wygenerowania nowego obiektu nie będzie wymagać przekazywania żadnych argumentów, jak to ma miejsce w implementacjach omawianego wzorca. Generator sam będzie wiedział, jak tworzyć nowe obiekty.

Zazwyczaj generator definiuje jedną tylko metodę, która wytwarza nowe obiekty. W naszym przypadku będzie to metoda `next()`; włączmy ją do swojego zestawu narzędzi:

```
//: net/mindview/util/Generator.java
// Interfejs uogólniony.
package net.mindview.util;
public interface Generator<T> { T next(); } ///:~
```

Typ wartości zwracanej przez metodę `next()` jest parametryzowany przez T. Jak widać, uogólnienia stosuje się w interfejsach bardzo podobnie jak w klasach.

Potrzebujemy jeszcze kilku klas ilustrujących implementację interfejsu `Generator`. Niech będzie to hierarchia klas opisujących gatunki kawy:

```
//: generics/coffee/Coffee.java
package generics.coffee;

public class Coffee {
  private static long counter = 0;
  private final long id = counter++;
  public String toString() {
    return getClass().getSimpleName() + " " + id;
  }
} ///:~

//: generics/coffee/Latte.java
package generics.coffee;
public class Latte extends Coffee {} ///:~

//: generics/coffee/Mocha.java
package generics.coffee;
public class Mocha extends Coffee {} ///:~

//: generics/coffee/Cappuccino.java
package generics.coffee;
public class Cappuccino extends Coffee {} ///:~

//: generics/coffee/Americano.java
package generics.coffee;
public class Americano extends Coffee {} ///:~

//: generics/coffee/Breve.java
package generics.coffee;
public class Breve extends Coffee {} ///:~
```

Teraz można zaimplementować generator Generator<Coffee>, który będzie wytwarzał obiekty losowo wybieranych typów Coffee:

```
//: generics/coffee/CoffeeGenerator.java
// Generowanie egzemplarzy różnych podtypów Coffee:
package generics.coffee;
import java.util.*;
import net.mindview.util.*;

public class CoffeeGenerator
implements Generator<Coffee>, Iterable<Coffee> {
  private Class[] types = { Latte.class, Mocha.class,
    Cappuccino.class, Americano.class, Breve.class, };
  private static Random rand = new Random(47);
  public CoffeeGenerator() {}
  // Dla potrzeb iteracji:
  private int size = 0;
  public CoffeeGenerator(int sz) { size = sz; }
  public Coffee next() {
    try {
      return (Coffee)
        types[rand.nextInt(types.length)].newInstance();
      // Zgłaszanie błędów programistycznych w czasie wykonania:
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  class CoffeeIterator implements Iterator<Coffee> {
    int count = size;
    public boolean hasNext() { return count > 0; }
    public Coffee next() {
      count--;
      return CoffeeGenerator.this.next();
    }
    public void remove() { // Bez implementacji
      throw new UnsupportedOperationException();
    }
  }
  public Iterator<Coffee> iterator() {
    return new CoffeeIterator();
  }
  public static void main(String[] args) {
    CoffeeGenerator gen = new CoffeeGenerator();
    for(int i = 0; i < 5; i++)
      System.out.println(gen.next());
    for(Coffee c : new CoffeeGenerator(5))
      System.out.println(c);
  }
} /* Output:
Americano 0
Latte 1
Americano 2
Mocha 3
Mocha 4
Breve 5
Americano 6
Latte 7
Cappuccino 8
Cappuccino 9
*///:~
```

Sparametryzowany interfejs `Generator` zapewnia zwracanie z metody `next()` egzemplarza zgodnego z typem parametryzującym. Klasa `CoffeeGenerator` implementuje do tego interfejs `Iterable`, co pozwala na wykorzystanie tej implementacji generatora również w pętlach forcach. Trzeba jednak pamiętać, że iteracja wymaga obecności „znacznika końca", tu występującego pod postacią licznika — jest on ustawiany przez drugą wersję konstruktora klasy.

A oto druga implementacja interfejsu `Generator<T>`, tym razem generująca elementy ciągu Fibonacciego:

```
//: generics/Fibonacci.java
// Generowanie ciągu Fibonacciego.
import net.mindview.util.*;

public class Fibonacci implements Generator<Integer> {
  private int count = 0;
  public Integer next() { return fib(count++); }
  private int fib(int n) {
    if(n < 2) return 1;
    return fib(n-2) + fib(n-1);
  }
  public static void main(String[] args) {
    Fibonacci gen = new Fibonacci();
    for(int i = 0; i < 18; i++)
      System.out.print(gen.next() + " ");
  }
} /* Output:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
*///:~
```

Choć w klasie i poza nią operujemy na wartościach typu `int`, parametr typowy został określony jako `Integer`. To skutek jednego z ograniczeń uogólnień w Javie: typy parametryzujące nie mogą być typami podstawowymi. Na szczęście dzięki wygodnemu mechanizmowi automatycznego pakowania takich wartości w obiekty (i automatycznego wypakowywania ich z tych obiektów) w Javie SE5 nie musimy kłopotać się jawną konwersją.

Możemy pójść o krok dalej i zaimplementować w generatorze ciągu Fibonacciego interfejs iteracji `Iterable`. Można w tym celu ponownie zaimplementować klasę z dodatkowym interfejsem, ale ta opcja zakłada pełną kontrolę nad pierwotną implementacją — inaczej może się okazać, że czynniki zewnętrzne zmuszają co i rusz do przepisywania rozszerzonej implementacji. Zamiast tego możemy uzupełnić klasę o interfejs za pomocą *adaptera* (kolejnego wzorca projektowego już prezentowanego w książce).

Adaptery można implementować rozmaicie. Można na przykład utworzyć zaadaptowaną klasę za pomocą dziedziczenia:

```
//: generics/IterableFibonacci.java
// Adaptacja klasy generatora ciągu Fibonacciego do interfejsu Iterable.
import java.util.*;

public class IterableFibonacci
extends Fibonacci implements Iterable<Integer> {
  private int n;
  public IterableFibonacci(int count) { n = count; }
```

```
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      public boolean hasNext() { return n > 0; }
      public Integer next() {
        n--;
        return IterableFibonacci.this.next();
      }
      public void remove() { // Bez implementacji
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    for(int i : new IterableFibonacci(18))
      System.out.print(i + " ");
  }
} /* Output:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
*///:~
```

Aby użyć obiektu klasy `IterableFibonacci` w pętli `foreach`, należy przekazać do konstruktora obiektu wartość graniczną ciągu — po jej osiągnięciu operacja przesunięcia iteratora zwróci `false` i tym samym przerwie pętlę.

Ćwiczenie 7. Zaadaptuj klasę `Fibonacci` do wymogów interfejsu `Iterable` na bazie kompozycji (a nie dziedziczenia) (2).

Ćwiczenie 8. Naśladując przykład z hierarchią `Coffee`, utwórz hierarchię bohaterów swojego ulubionego filmu (`StoryCharacter`) z podziałem na gałęzie `GoodGuys` (dobrzy) i `BadGuys` (źli). Wzorując się na klasie `CoffeeGenerator`, utwórz generator obiektów `StoryCharacter` (2).

Uogólnianie metod

Jak dotąd parametryzowaliśmy jedynie całe klasy. Okazuje się jednak, że można równie dobrze parametryzować pojedyncze metody w obrębie klasy. Sama klasa zawierająca takie metody może, ale nie musi być klasą uogólnioną — jej parametryzacja jest niezależna od posiadania metod uogólnionych.

Uogólnianie metod pozwala na różnicowanie zachowania metody w sposób niezależny od klasy. Możesz śmiało przyjąć wytyczną, że metody uogólnione powinieneś stosować „wszędzie, gdzie się da". To jest wszędzie tam, gdzie pożądany efekt można uzyskać uogólnieniem metody, a nie uogólnieniem całej klasy. Do tego w przypadku metod statycznych nie ma dostępu do parametrów uogólnienia klasy, więc aby skorzystać z zalet uogólniania, należy uogólnić również metody statyczne.

Aby zdefiniować metodę uogólnioną, wystarczy dołożyć listę parametrów typowych przed deklaracją typu wartości zwracanej metody, jak tutaj:

```
// generics/GenericMethods.java

public class GenericMethods {
  public <T> void f(T x) {
    System.out.println(x.getClass().getName());
  }
  public static void main(String[] args) {
    GenericMethods gm = new GenericMethods();
    gm.f("");
    gm.f(1);
    gm.f(1.0);
    gm.f(1.0F);
    gm.f('c');
    gm.f(gm);
  }
} /* Output:
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Float
java.lang.Character
GenericMethods
*///:~
```

Klasa `GenericMethods` nie została sparametryzowana, choć oczywiście parametryzacja może dotyczyć zarówno metod, jak i klasy równocześnie. W tym przypadku jedyną metodą z parametrem typowym jest metoda `f()`. Rozpoznaje się to po obecności listy parametrów typowych przed typem zwracanym metody.

Zauważ, że w przypadku klasy uogólnionej konkretyzacji parametrów typowych dokonuje się przy tworzeniu egzemplarza klasy. W przypadku metod uogólnionych zazwyczaj nie trzeba precyzować parametrów typowych, bo kompilator może je określić sam. To zachowanie kompilatora nosi nazwę *dedukcji typu argumentu* (ang. *argument type inference*). Wywołania metody `f()` wyglądają więc zupełnie zwyczajnie; samo uogólnienie metody dało zaś efekt przeciążenia tej metody dla niezliczonej ilości typów argumentów. Metoda taka może w wywołaniu przyjąć nawet obiekt klasy `GenericMethods`.

Wywołania `f()` z użyciem typów podstawowych angażują mechanizm automatycznego pakowania tych wartości w obiekty odpowiednich typów. Metody uogólnione w połączeniu z tym mechanizmem mogą posłużyć do wyeliminowania kodu, który wcześniej wymagał ręcznej konwersji.

Ćwiczenie 9. Zmodyfikuj program `GenericMethods.java` tak, aby metoda `f()` przyjmowała trzy argumenty, z których każdy jest potencjalnie innego typu (1).

Ćwiczenie 10. Zmodyfikuj poprzednie ćwiczenie tak, aby jeden z argumentów metody `f()` nie podlegał parametryzacji (1).

Wykorzystywanie dedukcji typu argumentu

Jedną z wad zarzucanych uogólnieniom jest wydłużanie i zmniejszanie czytelności kodu źródłowego poprzez wydłużenie określeń typów. Dobrym przykładem jest program *holding/MapOfList.java* z rozdziału „Kolekcje obiektów". Tworzenie kontenera `Map` zawierającego kontenery `List` wygląda tam tak:

```
Map<Person, List<? extends Pet>> petPeople =
  new HashMap<Person, List<? extends Pet>>();
```

(znaczenie znaków zapytania i słowa extends wyjaśnię w dalszej części rozdziału). Najwyraźniej trzeba się powtarzać, a przecież kompilator mógłby wywnioskować typy jednej listy parametrów na podstawie określonej w całości drugiej listy. Niestety, nie potrafi tego, ale rzecz można nieco uprościć, wykorzystując dedukcję typu argumentu w metodach uogólnionych. Jedną z możliwości jest utworzenie narzędzia zawierającego zestaw metod statycznych, tworzących najczęściej wykorzystywane konkretyzacje rozmaitych kontenerów:

```
//: net/mindview/util/New.java
// Narzędzia upraszczające tworzenie kontenerów uogólnionych
// wykorzystujące dedukcję typów argumentów.
package net.mindview.util;
import java.util.*;

public class New {
  public static <K,V> Map<K,V> map() {
    return new HashMap<K,V>();
  }
  public static <T> List<T> list() {
    return new ArrayList<T>();
  }
  public static <T> LinkedList<T> lList() {
    return new LinkedList<T>();
  }
  public static <T> Set<T> set() {
    return new HashSet<T>();
  }
  public static <T> Queue<T> queue() {
    return new LinkedList<T>();
  }
  // Przykłady:
  public static void main(String[] args) {
    Map<String, List<String>> sls = New.map();
    List<String> ls = New.list();
    LinkedList<String> lls = New.lList();
    Set<String> ss = New.set();
    Queue<String> qs = New.queue();
  }
} ///:~
```

W metodzie `main()` widać przykłady użycia proponowanych narzędzi: dedukcja typów argumentów eliminuje potrzebę powtarzania typów dla listy parametrów uogólnienia. Możemy zastosować to w ulepszonej wersji programu *holding/MapOfList.java*:

```
//: generics/SimplerPets.java
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;

public class SimplerPets {
  public static void main(String[] args) {
    Map<Person, List<? extends Pet>> petPeople = New.map();
    // Reszta kodu bez zmian...
  }
} ///:~
```

Przykład wykorzystania dedukcji typu argumentu jest niewątpliwie ciekawy, trudno jednak jednoznacznie docenić w nim przydatność tej techniki. Osoba czytająca kod musi poznać i ogarnąć dodatkową bibliotekę, więc być może mniejszym złem byłoby jednak — dla uproszczenia! — pozostawienie pierwotnego zapisu (z uciążliwymi, owszem, powtórzeniami). Ale gdyby prezentowane narzędzie znalazło się w standardowej bibliotece Javy, sens jego stosowania byłby już niepodważalny.

Dedukcja typów nie działa poza operacjami przypisania. Jeśli wynik wywołania metody `New.map()` zostanie wykorzystany w roli argumentu innej metody, kompilator nie będzie nawet podejmował próby dedukcji. Potraktuje wywołanie metody jako zamiar przypisania wartości zwracanej do obiektu klasy `Object`, jak tutaj:

```
//: generics/LimitsOfInference.java
import typeinfo.pets.*;
import java.util.*;

public class LimitsOfInference {
  static void
  f(Map<Person, List<? extends Pet>> petPeople) {}
  public static void main(String[] args) {
    // f(New.map()); // Nie skompiluje się
  }
} ///:~
```

Ćwiczenie 11. Przetestuj narzędzie *New.java*, tworząc własne klasy i sprawdzając, czy klasa `New` daje sobie z nimi radę (1).

Jawne określanie typu

Choć rzadko jest to potrzebne, typ konkretyzujący metodę uogólnioną można podawać również jawnie, za pomocą specjalnej składni. Zakłada ona umieszczenie nazwy typu w nawiasach kątowych za nazwą klasy i kropką, a jeszcze przed nazwą metody. W przypadku wywołania metody bieżącej klasy należy zastosować słowo `this` i kropkę; jeśli będzie to metoda statyczna, przed kropką należy umieścić nazwę klasy. Problem występujący w programie *LimitsOfInference.java* można więc wyeliminować tak:

```
//: generics/ExplicitTypeSpecification.java
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;

public class ExplicitTypeSpecification {
  static void f(Map<Person, List<Pet>> petPeople) {}
  public static void main(String[] args) {
    f(New.<Person, List<Pet>>map());
  }
} ///:~
```

Eliminuje to co prawda zalety klasy `New` związane z redukcją ilości kodu, który trzeba wpisać, ale pamiętaj, że owa dodatkowa składnia jest konieczna jedynie tam, gdzie z braku prostego przypisania kompilator nie może zastosować dedukcji typów argumentów.

Ćwiczenie 12. Powtórz poprzednie ćwiczenie z użyciem składni jawnego określania typów konkretyzacji uogólnienia (1).

Metody uogólnione ze zmiennymi listami argumentów

Metody uogólnione mogą posiadać zmienne listy argumentów:

```
//: generics/GenericVarargs.java
import java.util.*;

public class GenericVarargs {
  public static <T> List<T> makeList(T... args) {
    List<T> result = new ArrayList<T>();
    for(T item : args)
      result.add(item);
    return result;
  }
  public static void main(String[] args) {
    List<String> ls = makeList("A");
    System.out.println(ls);
    ls = makeList("A", "B", "C");
    System.out.println(ls);
    ls = makeList("ABCDEFFHIJKLMNOPQRSTUVWXYZ".split(""));
    System.out.println(ls);
  }
} /* Output:
[A]
[A, B, C]
[, A, B, C, D, E, F, F, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z]
*///:~
```

Metoda `makeList()` pełni tu rolę odpowiednika metody `java.util.Arrays.asList()` z biblioteki standardowej Javy.

Metoda uogólniona w służbie klasy Generator

Byłoby wygodnie, gdyby generator dało się wykorzystać do wypełnienia kolekcji; taką operację można „uogólnić":

```
//: generics/Generators.java
// Narzędzie współpracujące z klasami Generator.
import generics.coffee.*;
import java.util.*;
import net.mindview.util.*;

public class Generators {
  public static <T> Collection<T>
  fill(Collection<T> coll, Generator<T> gen, int n) {
    for(int i = 0; i < n; i++)
      coll.add(gen.next());
    return coll;
  }
  public static void main(String[] args) {
    Collection<Coffee> coffee = fill(
      new ArrayList<Coffee>(), new CoffeeGenerator(), 4);
    for(Coffee c : coffee)
      System.out.println(c);
    Collection<Integer> fnumbers = fill(
      new ArrayList<Integer>(), new Fibonacci(), 12);
```

```
    for(int i : fnumbers)
      System.out.print(i + ", ");
  }
} /* Output:
Americano 0
Latte 1
Americano 2
Mocha 3
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
*///:~
```

Zauważ, że uogólniona metoda `fill()` może być łatwo wykorzystana do wypełnienia kontenerów klasami `Coffee` i `Integer` z użyciem odpowiednich generatorów tych klas.

Ćwiczenie 13. Przeciąż metodę `fill()` tak, aby typy argumentów i wartości zwracanej były podtypami kolekcji: `List`, `Queue` i `Set`. W ten sposób zachowasz typ kontenera. Czy można przeciążyć metodę tak, aby rozróżniać `List` od `LinkedList` (4)?

Uniwersalny Generator

Poniżej prezentuję klasę tworzącą generator dla dowolnej klasy posiadającej konstruktor domyślny. W celu zmniejszenia ilości kodu zdecydowałem się na zdefiniowanie metody uogólnionej wytwarzającej generator `BasicGenerator`:

```
//: net/mindview/util/BasicGenerator.java
// Automatycznie tworzy generator dla klasy posiadającej
// konstruktor domyślny (bezargumentowy).
package net.mindview.util;

public class BasicGenerator<T> implements Generator<T> {
  private Class<T> type;
  public BasicGenerator(Class<T> type){ this.type = type; }
  public T next() {
    try {
      // Zakładamy, że typ jest klasą publiczną:
      return type.newInstance();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  // Utworzenie domyślnego generatora dla podanego typu:
  public static <T> Generator<T> create(Class<T> type) {
    return new BasicGenerator<T>(type);
  }
} ///:~
```

Klasa ta udostępnia podstawową implementację tworzącą obiekty klasy po pierwsze publicznej (bo generator `BasicGenerator` znajduje się w osobnym pakiecie, więc owa klasa musi mieć dostęp publiczny, a nie jedynie pakietowy), po drugie zaś posiadającej konstruktor domyślny (ten pozbawiony argumentów wywołania). Aby utworzyć egzemplarz generatora `BasicGenerator`, należy wywołać statyczną metodę `create()` i przekazać do niej nazwę typu (`*.class`). Uogólniona metoda `create()` pozwala na stosowanie zapisu `BasicGenerator.create(Typ.class)` zamiast udziwnionego `new BasicGenerator <Typ>(Typ.class)`.

Oto przykład prostej klasy wyposażonej w konstruktor domyślny:

```
//: generics/CountedObject.java

public class CountedObject {
  private static long counter = 0;
  private final long id = counter++;
  public long id() { return id; }
  public String toString() { return "CountedObject " + id;}
} ///:~
```

Klasa `CountedObject` zlicza utworzone egzemplarze tejże klasy i wypisuje je metodą `toString()`.

Za pomocą generatora `BasicGenerator` można łatwo utworzyć generator obiektów klasy `CountedObject`:

```
//: generics/BasicGeneratorDemo.java
import net.mindview.util.*;

public class BasicGeneratorDemo {
  public static void main(String[] args) {
    Generator<CountedObject> gen =
      BasicGenerator.create(CountedObject.class);
    for(int i = 0; i < 5; i++)
      System.out.println(gen.next());
  }
} /* Output:
CountedObject 0
CountedObject 1
CountedObject 2
CountedObject 3
CountedObject 4
*///:~
```

Widać, jak użycie metody uogólnionej zmniejsza ilość pisaniny potrzebnej normalnie do utworzenia obiektu generatora. Uogólnienia w Javie wymuszają tak czy inaczej przekazanie obiektu `Class`, więc można go śmiało użyć do dedukcji typu argumentu w metodzie `create()`.

Ćwiczenie 14. Zmodyfikuj program *BasicGeneratorDemo.java* tak, aby wykorzystywał jawną formę tworzenia generatora (czyli zamiast wywołania metody `create()` użyj jawnie odpowiedniego konstruktora) (1).

Upraszczanie stosowania krotek

Dedukcję typów argumentów konkretyzacji uogólnienia możemy wykorzystać w połączeniu z importem statycznym do uproszczenia stosowania krotek. Oto przykład tworzenia krotek za pośrednictwem statycznej metody przeciążonej:

```
//: net/mindview/util/Tuple.java
// Biblioteka krotek z dedukcją typów argumentów.
package net.mindview.util;

public class Tuple {
  public static <A,B> TwoTuple<A,B> tuple(A a, B b) {
```

```
    return new TwoTuple<A,B>(a, b);
  }
  public static <A,B,C> ThreeTuple<A,B,C>
  tuple(A a, B b, C c) {
    return new ThreeTuple<A,B,C>(a, b, c);
  }
  public static <A,B,C,D> FourTuple<A,B,C,D>
  tuple(A a, B b, C c, D d) {
    return new FourTuple<A,B,C,D>(a, b, c, d);
  }
  public static <A,B,C,D,E>
  FiveTuple<A,B,C,D,E> tuple(A a, B b, C c, D d, E e) {
    return new FiveTuple<A,B,C,D,E>(a, b, c, d, e);
  }
} ///:~
```

Takic metody pozwalają nam zmodyfikować program *TupleTest.java*:

```
//: generics/TupleTest2.java
import net.mindview.util.*;
import static net.mindview.util.Tuple.*;

public class TupleTest2 {
  static TwoTuple<String,Integer> f() {
    return tuple("hej", 47);
  }
  static TwoTuple f2() { return tuple("hej", 47); }
  static ThreeTuple<Amphibian,String,Integer> g() {
    return tuple(new Amphibian(), "hej", 47);
  }
  static
  FourTuple<Vehicle,Amphibian,String,Integer> h() {
    return tuple(new Vehicle(), new Amphibian(), "hej", 47);
  }
  static
  FiveTuple<Vehicle,Amphibian,String,Integer,Double> k() {
    return tuple(new Vehicle(), new Amphibian(),
      "hej", 47, 11.1);
  }
  public static void main(String[] args) {
    TwoTuple<String,Integer> ttsi = f();
    System.out.println(ttsi);
    System.out.println(f2());
    System.out.println(g());
    System.out.println(h());
    System.out.println(k());
  }
} /* Output: (80% match)
(hej, 47)
(hej, 47)
(Amphibian@7d772e, hej, 47)
(Vehicle@757aef, Amphibian@d9f9c3, hej, 47)
(Vehicle@1a46e30, Amphibian@3e25a5, hej, 47, 11.1)
*///:~
```

Zauważ, że metoda `f()` zwraca sparametryzowany obiekt `TwoTuple`, podczas gdy `f2()` zwraca obiekt tej samej klasy, ale niesparametryzowany. Kompilator nie oprotestuje wywołania `f2()`, bo wartość zwracana przez wywołanie nie jest w programie nigdzie

używana jako sparametryzowana; w pewnym sensie została poddana rzutowaniu w górę do niesparametryzowanej postaci TwoTuple, ale gdybyś spróbował przypisać wynik wywołania f2() do egzemplarza sparametryzowanej krotki TwoTuple, kompilator wystosowałby ostrzeżenie.

Ćwiczenie 15. Zweryfikuj powyższe twierdzenie (1).

Ćwiczenie 16. Dodaj do programu *Tuple.java* klasę SixTuple i przetestuj ją w programie *TupleTest2.java* (2).

Uniwersalny kontener Set

W ramach kolejnego przykładu zastosowań metod uogólnionych można rozważyć matematyczne relacje, które wyrażają kontenery Set (zbiory). Owe relacje można wygodnie definiować jako metody uogólnione, zdatne do stosowania z wieloma typami elementów zbiorów:

```
//: net/mindview/util/Sets.java
package net.mindview.util;
import java.util.*;

public class Sets {
  public static <T> Set<T> union(Set<T> a, Set<T> b) {
    Set<T> result = new HashSet<T>(a);
    result.addAll(b);
    return result;
  }
  public static <T>
  Set<T> intersection(Set<T> a, Set<T> b) {
    Set<T> result = new HashSet<T>(a);
    result.retainAll(b);
    return result;
  }
  // Odjęcie zbioru od nadzbioru:
  public static <T> Set<T>
  difference(Set<T> superset, Set<T> subset) {
    Set<T> result = new HashSet<T>(superset);
    result.removeAll(subset);
    return result;
  }
  // Relacja zwrotna -- wszystko spoza części wspólnej:
  public static <T> Set<T> complement(Set<T> a, Set<T> b) {
    return difference(union(a, b), intersection(a, b));
  }
} ///:~
```

Pierwsze trzy metody powielają pierwszy argument przez kopiowanie zawartych w nim referencji do nowego obiektu HashSet, a więc bez modyfikowania pierwotnego zbioru. Wartością zwracaną jest nowy obiekt Set.

Matematyczne operacje na zbiorach są reprezentowane czterema metodami: union() zwraca zbiór (Set) zawierający sumę obu argumentów, intersection() zwraca zbiór zawierający elementy wspólne dla obu argumentów, difference() zwraca różnicę zbioru superset-subset, a metoda complement() daje w wyniku zbiór Set stanowiący różnicę sumy obu

zbiorów i ich części wspólnej. Aby utworzyć prosty przykład ilustrujący działanie wymienionych metod, utworzymy typ wyliczeniowy z symbolicznymi nazwami różnych kolorów akwareli:

```
//: generics/watercolors/Watercolors.java
package generics.watercolors;

public enum Watercolors {
  ZINC, LEMON_YELLOW, MEDIUM_YELLOW, DEEP_YELLOW, ORANGE,
  BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER, VIOLET,
  CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
  COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE,
  SAP_GREEN, YELLOW_OCHRE, BURNT_SIENNA, RAW_UMBER,
  BURNT_UMBER, PAYNES_GRAY, IVORY_BLACK
} ///:~
```

Dla wygody (aby nie trzeba było kwalifikować wszystkich wartości zbioru wyliczeniowego nazwą jego typu) typ ten zostanie zaimportowany statycznie do następnego przykładu. Będzie on korzystał z kontenera EnumSet stanowiącego narzędzie (nowość w Javie SE5) do łatwego tworzenia zbiorów (Set), których elementami są wartości typu wyliczeniowego (o kontenerze EnumSet dowiesz się więcej w rozdziale „Typy wyliczeniowe"). Statyczna metoda EnumSet.range() otrzymuje w wywołaniu zakres wartości typu wyliczeniowego (określony pierwszym i ostatnim elementem zakresu) i tworzy zbiór ograniczony do tego zakresu:

```
//: generics/WatercolorSets.java
import generics.watercolors.*;
import java.util.*;
import static net.mindview.util.Print.*;
import static net.mindview.util.Sets.*;
import static generics.watercolors.Watercolors.*;

public class WatercolorSets {
  public static void main(String[] args) {
    Set<Watercolors> set1 =
      EnumSet.range(BRILLIANT_RED, VIRIDIAN_HUE);
    Set<Watercolors> set2 =
      EnumSet.range(CERULEAN_BLUE_HUE, BURNT_UMBER);
    print("set1: " + set1);
    print("set2: " + set2);
    print("union(set1, set2): " + union(set1, set2));
    Set<Watercolors> subset = intersection(set1, set2);
    print("intersection(set1, set2): " + subset);
    print("difference(set1, subset): " +
      difference(set1, subset));
    print("difference(set2, subset): " +
      difference(set2, subset));
    print("complement(set1, set2): " +
      complement(set1, set2));
  }
} /* Output: (Sample)
set1: [BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER, VIOLET, CERULEAN_BLUE_HUE,
PHTHALO_BLUE, ULTRAMARINE, COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE]
set2: [CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE, COBALT_BLUE_HUE,
PERMANENT_GREEN, VIRIDIAN_HUE, SAP_GREEN, YELLOW_OCHRE, BURNT_SIENNA,
RAW_UMBER, BURNT_UMBER]
```

```
union(set1, set2): [RAW_UMBER, ROSE_MADDER, BURNT_UMBER, SAP_GREEN, BURNT_SIENNA,
PHTHALO_BLUE, VIRIDIAN_HUE, CERULEAN_BLUE_HUE, YELLOW_OCHRE, VIOLET, MAGENTA,
COBALT_BLUE_HUE, PERMANENT_GREEN, BRILLIANT_RED, CRIMSON, ULTRAMARINE]
intersection(set1, set2): [PERMANENT_GREEN, COBALT_BLUE_HUE, VIRIDIAN_HUE,
PHTHALO_BLUE, CERULEAN_BLUE_HUE, ULTRAMARINE]
difference(set1, subset): [ROSE_MADDER, VIOLET, MAGENTA, CRIMSON, BRILLIANT_RED]
difference(set2, subset): [RAW_UMBER, BURNT_UMBER, SAP_GREEN, BURNT_SIENNA,
YELLOW_OCHRE]
complement(set1, set2): [RAW_UMBER, ROSE_MADDER, BURNT_UMBER, SAP_GREEN,
BURNT_SIENNA, YELLOW_OCHRE, VIOLET, MAGENTA, CRIMSON, BRILLIANT_RED]
*///:~
```

Wyniki poszczególnych operacji można przeanalizować na wyjściu programu.

Następny program wykorzysta metodę `Sets.difference()` do zilustrowania różnic pomiędzy różnymi klasami hierarchii `Collection` i `Map` dostępnymi w bibliotece `java.util`:

```
//: net/mindview/util/ContainerMethodDifferences.java
package net.mindview.util;
import java.lang.reflect.*;
import java.util.*;

public class ContainerMethodDifferences {
  static Set<String> methodSet(Class<?> type) {
    Set<String> result = new TreeSet<String>();
    for(Method m : type.getMethods())
      result.add(m.getName());
    return result;
  }
  static void interfaces(Class<?> type) {
    System.out.print("Interfejsy w " +
      type.getSimpleName() + ": ");
    List<String> result = new ArrayList<String>();
    for(Class<?> c : type.getInterfaces())
      result.add(c.getSimpleName());
    System.out.println(result);
  }
  static Set<String> object = methodSet(Object.class);
  static { object.add("klon"); }
  static void
  difference(Class<?> superset, Class<?> subset) {
    System.out.print(superset.getSimpleName() +
      " dziedziczy po " + subset.getSimpleName() + ", dodając: ");
    Set<String> comp = Sets.difference(
      methodSet(superset), methodSet(subset));
    comp.removeAll(object); // Nie pokazujemy metod klasy 'Object'
    System.out.println(comp);
    interfaces(superset);
  }
  public static void main(String[] args) {
    System.out.println("Collection: " +
      methodSet(Collection.class));
    interfaces(Collection.class);
    difference(Set.class, Collection.class);
    difference(HashSet.class, Set.class);
    difference(LinkedHashSet.class, HashSet.class);
    difference(TreeSet.class, Set.class);
```

```
    difference(List.class, Collection.class);
    difference(ArrayList.class, List.class);
    difference(LinkedList.class, List.class);
    difference(Queue.class, Collection.class);
    difference(PriorityQueue.class, Queue.class);
    System.out.println("Map: " + methodSet(Map.class));
    difference(HashMap.class, Map.class);
    difference(LinkedHashMap.class, HashMap.class);
    difference(SortedMap.class, Map.class);
    difference(TreeMap.class, Map.class);
  }
} ///:~
```

Wyjście powyższego programu zostało wykorzystane w podrozdziale „Podsumowanie" rozdziału „Kolekcje obiektów".

Ćwiczenie 17. Przestudiuj dokumentację JDK dla klasy `EnumSet`. Znajdziesz tam metodę `clone()`. Ale nie możesz wykorzystać metody `clone()` do wykonania „klonu" referencji `Set` przekazywanego w *Sets.java*. Czy można zmodyfikować program *Sets.java* tak, żeby obsługiwał zarówno przypadek ogólny interfejsu `Set`, jak i przypadek specjalny interfejsu `EnumSet` z użyciem metody `clone()` do tworzenia nowych kontenerów `HashSet` (4)?

Anonimowe klasy wewnętrzne

Uogólnienia można wykorzystywać również w klasach wewnętrznych i anonimowych klasach wewnętrznych. Oto przykład implementujący interfejs `Generator` za pośrednictwem anonimowych klas wewnętrznych:

```
// generics/BankTeller.java
// Bardzo prosta symulacja okienka kasowego.
import java.util.*;
import net.mindview.util.*;

class Customer {
  private static long counter = 1;
  private final long id = counter++;
  private Customer() {}
  public String toString() { return "Klient " + id; }
  // Metoda tworząca obiekty klasy Generator:
  public static Generator<Customer> generator() {
    return new Generator<Customer>() {
      public Customer next() { return new Customer(); }
    };
  }
}

class Teller {
  private static long counter = 1;
  private final long id = counter++;
  private Teller() {}
  public String toString() { return "Kasjer " + id; }
  // Pojedynczy obiekt generatora Generator:
```

```
  public static Generator<Teller> generator =
    new Generator<Teller>() {
      public Teller next() { return new Teller(); }
    };
}

public class BankTeller {
  public static void serve(Teller t, Customer c) {
    System.out.println(t + " obsługuje klienta " + c);
  }
  public static void main(String[] args) {
    Random rand = new Random(47);
    Queue<Customer> line = new LinkedList<Customer>();
    Generators.fill(line, Customer.generator(), 15);
    List<Teller> tellers = new ArrayList<Teller>();
    Generators.fill(tellers, Teller.generator, 4);
    for(Customer c : line)
      serve(tellers.get(rand.nextInt(tellers.size())), c);
  }
} /* Output:
Kasjer 3 obsługuje klienta Klient 1
Kasjer 2 obsługuje klienta Klient 2
Kasjer 3 obsługuje klienta Klient 3
Kasjer 1 obsługuje klienta Klient 4
Kasjer 1 obsługuje klienta Klient 5
Kasjer 3 obsługuje klienta Klient 6
Kasjer 1 obsługuje klienta Klient 7
Kasjer 2 obsługuje klienta Klient 8
Kasjer 3 obsługuje klienta Klient 9
Kasjer 3 obsługuje klienta Klient 10
Kasjer 2 obsługuje klienta Klient 11
Kasjer 4 obsługuje klienta Klient 12
Kasjer 2 obsługuje klienta Klient 13
Kasjer 1 obsługuje klienta Klient 14
Kasjer 1 obsługuje klienta Klient 15
*///:~
```

Klasy `Customer` i `Teller` posiadają prywatne konstruktory wymuszające użycie obiektów-generatorów. Klasa `Customer` posiada metodę `generator()` tworzącą w ramach każdego wywołania nowy egzemplarz `Generator<Customer>`. Z kolei klasa `Teller` udostępnia za pośrednictwem pola `generator` pojedynczy, publiczny obiekt generatora `Generator<Teller>`. Oba generatory są następnie wykorzystywane w wywołaniu metody `fill()` wewnątrz metody `main()`.

Ponieważ metoda `generator()` w klasie `Customer` i obiekt `Generator` w klasie `Teller` są składowymi statycznymi, nie mogą stanowić części interfejsu, nie ma więc możliwości uogólnienia takiego idiomu. Mimo to całość działa zupełnie nieźle z metodą `fill()`.

Pozostałym aspektom tego rodzaju symulacji przyjrzymy się w rozdziale „Współbieżność".

Ćwiczenie 18. Naśladując program *BankTeller.java*, napisz przykład, w którym wielkie ryby (`BigFish`) pochłaniają mniejsze (`LittleFish`) w oceanie (`Ocean`) (3).

Budowanie modeli złożonych

Ważną zaletą uogólnień jest możliwość upraszczania i bezpiecznego tworzenia modeli złożonych. Weźmy za przykład listę (List) krotek:

```
//: generics/TupleList.java
// Łączenie typów uogólnionych w złożonych typach uogólnionych.
import java.util.*;
import net.mindview.util.*;

public class TupleList<A,B,C,D>
extends ArrayList<FourTuple<A,B,C,D>> {
  public static void main(String[] args) {
    TupleList<Vehicle, Amphibian, String, Integer> tl =
      new TupleList<Vehicle, Amphibian, String, Integer>();
    tl.add(TupleTest.h());
    tl.add(TupleTest.h());
    for(FourTuple<Vehicle,Amphibian,String,Integer> i: tl)
      System.out.println(i);
  }
} /* Output: (75% match)
(Vehicle@11b86e7, Amphibian@35ce36, hej, 47)
(Vehicle@757aef, Amphibian@d9f9c3, hej, 47)
*///:~
```

Mimo pewnej rozwlekłości (zwłaszcza przy tworzeniu iteratora) otrzymaliśmy strukturę danych o całkiem pokaźnych możliwościach, przy mimo wszystko niewielkiej ilości kodu.

Kolejny przykład również będzie pokazywał łatwość budowania skomplikowanych modeli za pośrednictwem typów uogólnionych. W przykładzie mamy zestaw klas stanowiących cegiełki tego modelu; sam model opisuje punkt sprzedaży detalicznej z regałami, półkami i towarami:

```
//: generics/Store.java
// Składanie złożonego modelu z kontenerów uogólnionych.
import java.util.*;
import net.mindview.util.*;

class Product {
  private final int id;
  private String description;
  private double price;
  public Product(int IDnumber, String descr, double price){
    id = IDnumber;
    description = descr;
    this.price = price;
    System.out.println(toString());
  }
  public String toString() {
    return id + ": " + description + ", cena: " + price + " zł";
  }
  public void priceChange(double change) {
    price += change;
  }
  public static Generator<Product> generator =
```

```
    new Generator<Product>() {
      private Random rand = new Random(47);
      public Product next() {
        return new Product(rand.nextInt(1000), "Test",
          Math.round(rand.nextDouble() * 1000.0) + 0.99);
      }
    };
}

class Shelf extends ArrayList<Product> {
  public Shelf(int nProducts) {
    Generators.fill(this, Product.generator, nProducts);
  }
}

class Aisle extends ArrayList<Shelf> {
  public Aisle(int nShelves, int nProducts) {
    for(int i = 0; i < nShelves; i++)
      add(new Shelf(nProducts));
  }
}

class CheckoutStand {}
class Office {}

public class Store extends ArrayList<Aisle> {
  private ArrayList<CheckoutStand> checkouts =
    new ArrayList<CheckoutStand>();
  private Office office = new Office();
  public Store(int nAisles, int nShelves, int nProducts) {
    for(int i = 0; i < nAisles; i++)
      add(new Aisle(nShelves, nProducts));
  }
  public String toString() {
    StringBuilder result = new StringBuilder();
    for(Aisle a : this)
      for(Shelf s : a)
        for(Product p : s) {
          result.append(p);
          result.append("\n");
        }
    return result.toString();
  }
  public static void main(String[] args) {
    System.out.println(new Store(14, 5, 10));
  }
} /* Output:
258: Test, cena: 400.99 zł
861: Test, cena: 160.99 zł
868: Test, cena: 417.99 zł
207: Test, cena: 268.99 zł
551: Test, cena: 114.99 zł
278: Test, cena: 804.99 zł
520: Test, cena: 554.99 zł
140: Test, cena: 530.99 zł
...
*///:~
```

Efekt wywołania metody `Store.toString()` uwidacznia wielopoziomowe złożenie kontenerów, które mimo całej złożoności jest zabezpieczane ze strony systemu kontroli typów i daje się stosunkowo łatwo zarządzać. A i samo zmontowanie modelu nie było szczególnym wyzwaniem intelektualnym.

Ćwiczenie 19. Wzorując się na programie *Store.java*, zaproponuj model statku kontenerowca (2).

Tajemnica zacierania

Zagłębiając się w uogólnienia, zauważa się rzeczy, które pozornie nie mają sensu. Na przykład, choć można zapisać `ArrayList.class`, nie można zapisać `ArrayList<Integer>.class`. Spójrz też tutaj:

```
//: generics/ErasedTypeEquivalence.java
import java.util.*;

public class ErasedTypeEquivalence {
  public static void main(String[] args) {
    Class c1 = new ArrayList<String>().getClass();
    Class c2 = new ArrayList<Integer>().getClass();
    System.out.println(c1 == c2);
  }
} /* Output:
true
*///:~
```

Typy `ArrayList<String>` i `ArrayList<Integer>` można w uprawniony sposób uznać za różne. Typy różnicuje ich zachowanie, a umieszczenie wartości typu `Integer` w kontenerze `ArrayList<String>` spowoduje z pewnością inne zachowanie (mianowicie błąd) niż umieszczenie tej samej wartości w kontenerze `ArrayList<Integer>`. A mimo to powyższy program sugeruje, że te typy są identyczne.

Oto kolejny element układanki:

```
//: generics/LostInformation.java
import java.util.*;

class Frob {}
class Fnorkle {}
class Quark<Q> {}
class Particle<POSITION,MOMENTUM> {}

public class LostInformation {
  public static void main(String[] args) {
    List<Frob> list = new ArrayList<Frob>();
    Map<Frob,Fnorkle> map = new HashMap<Frob,Fnorkle>();
    Quark<Fnorkle> quark = new Quark<Fnorkle>();
    Particle<Long,Double> p = new Particle<Long,Double>();
    System.out.println(Arrays.toString(
      list.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      map.getClass().getTypeParameters()));
```

```
    System.out.println(Arrays.toString(
      quark.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      p.getClass().getTypeParameters()));
  }
} /* Output:
[E]
[K, V]
[Q]
[POSITION, MOMENTUM]
*///:~
```

Według dokumentacji JDK `Class.getTypeParameters()` „zwraca tablicę obiektów `TypeVariable` reprezentujących zmienne typów występujące w deklaracji uogólnienia (...)". Czyli niby powinniśmy mieć możliwość wykrycia wartości parametrów typowych, ale wyjście programu przekonuje, że w ten sposób można odszukać jedynie identyfikatory składające się na listę parametrów typowych, które interesują nas znacznie mniej.

Smutna prawda jest taka:

> *W kodzie uogólnionym nie ma dostępu do informacji o typach konkretyzujących uogólnienie.*

Można się więc dowiedzieć, jaki jest identyfikator reprezentujący parametr typowy, ale nie można nijak się dowiedzieć, jaki typ został użyty do konkretyzacji tego egzemplarza typu uogólnionego. Ten fakt szczególnie frustruje programistów przywykłych do C++ i stanowi źródło największych problemów przy pracy z uogólnieniami w Javie.

Otóż uogólnienia w Javie są implementowane na bazie *zacierania* (ang. *erasure*). Należy to rozumieć jako wymazywanie wszelkich informacji o typie konkretyzującym. W obrębie typu uogólnionego jedyną wiadomą jest to, że używamy obiektu. Dlatego też `List<String>` i `List<Integer>` w istocie *są* w czasie wykonania tymi samymi typami. Oba zostaną „obrane" z informacji o typach konkretyzujących do postaci *„surowego" typu* uogólnienia. Ogarnięcie kwestii zacierania i sposobu radzenia sobie z tym zjawiskiem jest jednym z większych wyzwań w opanowaniu uogólnień w Javie. Dlatego też tym aspektom uogólnień przyjrzymy się szczególnie uważnie.

Jak to się robi w C++

Oto przykład wykorzystujący odpowiedniki uogólnień w C++ noszące tam miano *szablonów*. Zwróć uwagę na podobieństwa składni wynikające zresztą z wzorowania się projektantów Javy na C++:

```
//: generics/Templates.cpp
#include <iostream>
using namespace std;

template<class T> class Manipulator {
  T obj;
public:
  Manipulator(T x) { obj = x; }
  void manipulate() { obj.f(); }
};
```

```
class HasF {
public:
  void f() { cout << "HasF::f()" << endl; }
};

int main() {
  HasF hf;
  Manipulator<HasF> manipulator(hf);
  manipulator.manipulate();
} /* Output:
HasF::f()
///:~
```

Obiekt klasy `Manipulator` przechowuje w sobie obiekt typu `T`. Interesuje nas metoda `manipulate()` wywołująca na rzecz tego obiektu metodę `f()`. Skąd wiadomo, że metoda `f()` jest dostępna dla typu reprezentowanego parametrem typowym `T`? Otóż jest to sprawdzane przez kompilator na etapie konkretyzacji szablonu — w miejscu utworzenia egzemplarza `Manipulator<HasF>` kompilator widzi, że typ `HasF` posiada metodę `f()`. Gdyby było inaczej, doszłoby do błędu kompilacji — w ten sposób objawia się kontrola typów w szablonach C++.

Pisanie tego rodzaju kodu w C++ jest proste, bo w momencie konkretyzacji szablonu kod szablonu zna typy wykorzystane do konkretyzacji. W Javie jest inaczej. Oto wersja typu `HasF` w języku Java:

```
//: generics/HasF.java

public class HasF {
  public void f() { System.out.println("HasF.f()"); }
} ///:~
```

Gdybyśmy na język Java chcieli przetłumaczyć resztę przykładu, wyglądałoby to tak:

```
//: generics/Manipulation.java
// {CompileTimeError} (Nie skompiluje się)

class Manipulator<T> {
  private T obj;
  public Manipulator(T x) { obj = x; }
  // Błąd (cannot find symbol: method f()):
  public void manipulate() { obj.f(); }
}

public class Manipulation {
  public static void main(String[] args) {
    HasF hf = new HasF();
    Manipulator<HasF> manipulator =
      new Manipulator<HasF>(hf);
    manipulator.manipulate();
  }
} ///:~
```

Z powodu mechanizmu zacierania kompilator nie potrafi skojarzyć wymagania, aby `manipulate()` mogła wywoływać `f()` na rzecz obicktu `obj`, z faktem, że `HasF` posiada metodę `f()`. Musimy mu asystować, nadając klasie uogólnionej *ramy uogólnienia* (ang. *bounds*),

instruujące kompilator odnośnie akceptowalnych typów. Rolę tę spełnia słowo kluczowe extends. Dzięki określeniu ramy uogólnienia poniższy kod się skompiluje:

```
//: generics/Manipulator2.java

class Manipulator2<T extends HasF> {
  private T obj;
  public Manipulator2(T x) { obj = x; }
  public void manipulate() { obj.f(); }
} ///:~
```

Rama <T extends HasF> mówi, że T musi być typu HasF albo dowolnego typu wyprowadzonego z HasF. Jeśli ten warunek jest spełniony, można bezpiecznie wywołać f() na rzecz obiektu obj.

Mówimy, że parametr typowy uogólnienia podlega *zatarciu do pierwszej ramy* (bo ram można określić więcej — o tym za chwilę) parametru typowego. Mówimy też o *zatarciu parametru typowego*. Kompilator zastępuje parametr typowy jego zatarciem, więc w powyższym przypadku T jest zacierany do postaci HasF, co daje efekt identyczny z podstawieniem HasF w miejsce T w całym ciele definicji klasy.

Być może zauważyłeś, że w programie *Manipulation2.java* uogólnienie niczemu nie służy. Równie dobrze moglibyśmy sami dokonać zatarcia i utworzyć klasę bez uogólnienia:

```
//: generics/Manipulator3.java

class Manipulator3 {
  private HasF obj;
  public Manipulator3(HasF x) { obj = x; }
  public void manipulate() { obj.f(); }
} ///:~
```

Dochodzimy niniejszym do ważnego wniosku: uogólnienia są przydatne jedynie wtedy, kiedy chcemy użyć parametrów typowych „ogólniejszych" od pewnego typu konkretnego (i jego typów pochodnych) — czyli tam, gdzie kod ma działać z różnymi klasami. W efekcie parametry typowe i ich zastosowania w użytecznym kodzie uogólnionym będą bardziej złożone niż zwykłe podstawienia klas. Nie można jednak stwierdzić, że wszystko w postaci <T extends HasF> jest wadliwe. Na przykład, jeśli klasa posiada metodę zwracającą T, uogólnienia są użyteczne, bo pozwolą na zwrócenie dokładnego typu:

```
//: generics/ReturnGenericType.java

class ReturnGenericType<T extends HasF> {
  private T obj;
  public ReturnGenericType(T x) { obj = x; }
  public T get() { return obj; }
} ///:~
```

Każdorazowo należy więc przyjrzeć się złożoności kodu, aby ocenić, czy uzasadnia zastosowanie uogólnień.

Do tematu ram wrócimy w dalszej części rozdziału.

Ćwiczenie 20. Utwórz interfejs z dwiema metodami i klasę, która implementuje ten interfejs i ze swej strony dodaje kolejną metodę. W innej klasie utwórz metodę uogólnioną z typem parametryzującym ograniczonym przez interfejs; pokaż, że wewnątrz tej metody uogólnionej można wywoływać metody interfejsu. W metodzie `main()` przekaż egzemplarz klasy implementującej interfejs do metody uogólnionej (1).

Słowo o zgodności migracji

Aby zlikwidować wszelkie potencjalne nieporozumienia co do natury zacierania, trzeba zrozumieć, że *nie* jest to cecha języka jako takiego. To jedynie kompromis w zakresie implementacji uogólnień w języku Java, którego konieczność wynika z pierwotnego braku przystosowania języka do typów ogólnych. Kompromis ten bywa kłopotliwy i tym ważniejsze jest zrozumienie przyczyn jego obecności.

Gdyby uogólnienia były obecne w Javie od wersji 1.0, z pewnością obywałyby się bez zacierania — jego miejsce zajmowałoby uszczegóławianie zachowujące typy konkretyzujące uogólnienie jako pełnoprawne jednostki programu, które można by poddać zwykłym operacjom języka z kontrolą typów, a także wykorzystywać w refleksji. Przekonasz się wkrótce, że zacieranie zmniejsza „ogólność" uogólnień. Mimo to pozostają one przydatne, choć mogłyby być jeszcze przydatniejsze.

W implementacji bazującej na zacieraniu typy uogólnione są traktowane jako typy drugiej kategorii, których nie można wykorzystywać w wielu ważnych kontekstach. Typy uogólnione są obecne jedynie na etapie statycznej kontroli typów, po czym każdy typ uogólniony programu jest zacierany przez wstawienie w jego miejsce zatarcia do ramy. Na przykład typ `List<T>` jest zastępowany przez zwyczajny List, a zwykłe zmienne typów są zacierane do typu `Object` (ale tylko w przypadku braku ramy).

Uzasadnieniem takiej implementacji jest umożliwienie wykorzystywania typów uogólnionych w bibliotekach nieuogólnionych i odwrotnie. Możliwość tę określa się często mianem *zgodności migracji* (ang. *migration compatibility*). W idealnym świecie przyszedłby kiedyś taki dzień, w którym wszystko zostałoby elegancko uogólnione, ale w praktyce, nawet jeśli programiści sami piszą wyłącznie kod uogólniony, muszą korzystać z bibliotek nieprzystosowanych do uogólnień, napisanych w języku Java wedle jego specyfiki sprzed wydania SE5. Autorzy tych bibliotek być może nigdy nie przekonają się do zalet uogólnień, a w najlepszym przypadku ich wdrożenie zajmie im jakiś czas.

Dlatego uogólnienia w Javie muszą zapewniać *zgodność wstecz* — istniejący kod i pliki klas nie mogą być przekreślone i muszą wciąż oznaczać to, co oznaczały wcześniej — ale równocześnie muszą dawać zgodność migracji, tak aby można było uogólniać biblioteki bez zrywania zgodności z kodem wykorzystującym te biblioteki w sposób nieuogólniony. Przy tak nakreślonych zadaniach projektanci Javy, przy udziale różnych grup pracujących nad rozwiązaniem, ustalili jedyne wykonalne rozwiązanie właśnie w postaci zacierania. Zacieranie umożliwia przechodzenie na typy uogólnione bez unieważniania istniejącego kodu nieuogólnionego.

Załóżmy na przykład, że nasza aplikacja wykorzystuje dwie biblioteki zewnętrzne: `X` i `Y`, oraz że `Y` wykorzystuje z kolei bibliotekę `Z`. Z dniem wydania wersji SE5 języka twórcy aplikacji i wymienionych bibliotek zaczną zapewne przechodzenie na uogólnienia.

Każdy z nich będzie jednak podejmował decyzje samodzielnie i będzie podlegał rozmaitym ograniczeniom, które zróżnicują termin i zakres uogólniania. Aby zapewnić zgodność migracji wszystkie zaangażowane w aplikacji biblioteki muszą być niezależne od ewentualnego użycia uogólnień w pozostałych bibliotekach (i samej aplikacji). Skoro tak, to nie powinny nawet być w stanie wykryć, że pozostałe biblioteki wykorzystują uogólnienia bądź ich jeszcze nie wykorzystują. Z tego zaś wynika, że fakt stosowania uogólnień w danej bibliotece musi zostać „zatarty".

Gdyby nie ten margines migracji, wszystkie biblioteki napisane w przeszłości straciłyby dla programistów przerzucających się na uogólnienia (a ten trend trudno byłoby zatrzymać) wszelką użyteczność. Tymczasem to również biblioteki stanowią o sile i możliwościach języka i to one decydują o produktywności programisty — odcięcie go od nich stanowiłoby koszt nie do przyjęcia. Czas pokaże, czy nie dałoby się zapewnić zgodności migracji inaczej.

Kłopotliwość zacierania

Ustaliliśmy już, że głównym uzasadnieniem dla zacierania jest zapewnienie możliwości płynnego przechodzenia od kodu uogólnionego do nieuogólnionego, a także wcielenie uogólnień do języka programowania bez zrywania zgodności z istniejącymi (jakże ważnymi) bibliotekami. Zacieranie umożliwia programiście-klientowi stosowanie w jego nieuogólnionym kodzie bibliotek wykorzystujących uogólnienia do czasu, kiedy ten programista sam będzie gotowy do użycia ich w swoich aplikacjach. Niechęć do zrywania zgodności kodu jest z pewnością szlachetną motywacją.

Jednakże zacieranie niesie ze sobą poważny koszt. Otóż typy uogólnione nie mogą uczestniczyć w operacji, która jawnie odwołuje się do ich typów w czasie wykonania, jak to ma miejsce choćby w przypadku operacji `instanceof` czy w wyrażeniach `new`. Utrata wszelkich informacji o typach konkretyzujących uogólnienie wymaga stałej samokontroli: trzeba wciąż przypominać sobie, że dostępność informacji o typie konkretyzującym za pośrednictwem parametru typu jest jedynie *pozorem*. Jeśli napiszesz coś takiego:

```
class Foo<T> {
  T var;
}
```

to przy tworzeniu egzemplarza `Foo`:

```
Foo<Cat> f = new Foo<Cat>();
```

kod klasy `Foo` jedynie pozornie otrzymuje informację, że operuje na typie `Cat`. Składnia sugeruje, że w całym ciele klasy dochodzi do zastąpienia parametru `T` typem `Cat`. Ale to tylko pozór i trzeba upominać samego siebie, że `T` w obrębie kodu tej klasy to „po prostu `Object`".

Dalej, zgodność migracji i zacieranie oznaczają, że użycie uogólnienia nie jest nigdy narzucane, nawet tam, gdzie byłoby to pożądane:

```
//: generics/ErasureAndInheritance.java

class GenericBase<T> {
  private T element;
```

```
  public void set(T arg) { arg = element; }
  public T get() { return element; }
}

class Derived1<T> extends GenericBase<T> {}

class Derived2 extends GenericBase {} // Bez ostrzeżenia

// class Derived3 extends GenericBase<?> {}
// Dziwny błąd:
// unexpected type found : ?
// required: class or interface without bounds

public class ErasureAndInheritance {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    Derived2 d2 = new Derived2();
    Object obj = d2.get();
    d2.set(obj); // Ostrzeżenie!
  }
} ///:~
```

Klasa `Derived2` dziedziczy po klasie uogólnionej `GenericBase`, ale nie określa typów konkretyzujących uogólnienie, a mimo to kompilator nie zgłasza nawet ostrzeżenia. Ostrzeżenie nie pojawi się dopóty, dopóki nie dojdzie do wywołania `set()`.

Do tłumienia komunikatów ostrzeżeń wykorzystuje się adnotację widoczną na powyższym listingu (adnotacja ta nie była rozpoznawana we wczesnych wersjach Javy SE5):

```
@SuppressWarnings("unchecked")
```

Zauważ, że adnotacją opatruje się nie całą klasę, a jedynie metodę, która miałaby spowodować ostrzeżenie. Przy tłumieniu ostrzeżeń należy zachować wstrzemięźliwość i jak najściślej zawężać zakres tłumienia, aby przypadkiem przez zbyt beztroskie tłumienie ostrzeżeń nie doprowadzić do ukrycia komunikatów o faktycznych problemach.

Błąd powodowany przez klasę `Derived3` oznacza niechybnie, że kompilator oczekuje surowej klasy bazowej.

Do tego wszystkiego trzeba dodać dodatkowy wysiłek związany z zarządzaniem ramami typów wszędzie tam, gdzie parametr typowy ma być interpretowany precyzyjniej niż tylko jako `Object`; wysiłku tego jest więcej niż przy stosowaniu typów sparametryzowanych w językach takich jak C++, Ada czy Eiffel, przy daleko mniejszych korzyściach. Nie należy na tej podstawie wnosić, że wymienione języki stanowią bezwzględnie lepsze zaplecze do rozwiązywania problemów programistycznych niż Java, nie sposób się jednak nie zgodzić, że mechanizmy parametryzacji typów są w nich daleko bardziej elastyczne i efektywne.

Na krawędzi

Z racji zacierania za najbardziej mylący aspekt uogólnień w Javie uważam możliwość reprezentowania rzeczy, które nie mają znaczenia. Oto przykład:

```
//: generics/ArrayMaker.java
import java.lang.reflect.*;
import java.util.*;

public class ArrayMaker<T> {
  private Class<T> kind;
  public ArrayMaker(Class<T> kind) { this.kind = kind; }
  @SuppressWarnings("unchecked")
  T[] create(int size) {
    return (T[])Array.newInstance(kind, size);
  }
  public static void main(String[] args) {
    ArrayMaker<String> stringMaker =
      new ArrayMaker<String>(String.class);
    String[] stringArray = stringMaker.create(9);
    System.out.println(Arrays.toString(stringArray));
  }
} /* Output:
[null, null, null, null, null, null, null, null, null]
*///:~
```

Choć obiekt `kind` jest deklarowany jako egzemplarz `Class<T>`, zacieranie powoduje, że jawi się po prostu jako egzemplarz `Class` bez parametru typowego. Kiedy więc wykorzystujemy go w operacjach, na przykład przy operacji tworzenia tablicy, wywołanie `Array.newInstance()` nie dysponuje informacją o szczegółowym typie `kind`; nie może więc wytworzyć tablicy elementów owego szczegółowego typu. Musimy się uciec do rzutowania, co z kolei powoduje pojawienie się ostrzeżenia, którego nie sposób uniknąć inaczej, niż przez stłumienie.

Zwróć uwagę na to, że wywołanie `Array.newInstance()` jest zalecanym sposobem tworzenia tablic typów uogólnionych.

Gdybyśmy zamiast tablicy tworzyli kontener, sprawy miałyby się zgoła inaczej:

```
//: generics/ListMaker.java
import java.util.*;

public class ListMaker<T> {
  List<T> create() { return new ArrayList<T>(); }
  public static void main(String[] args) {
    ListMaker<String> stringMaker= new ListMaker<String>();
    List<String> stringList = stringMaker.create();
  }
} ///:~
```

Kompilator nie wypisuje żadnych ostrzeżeń, choć wiemy, że (z powodu zacierania) w wyrażeniu `new ArrayList<T>()` w metodzie `create()` człon `<T>` zanika — w czasie wykonania w klasie nie ma żadnego `<T>`. Ale jeśli sami, wnioskując jak wyżej, zmienimy wyrażenie na `new ArrayList()`, kompilator wystosuje ostrzeżenie.

Czy obecność bądź brak `<T>` jest znacząca? A co w sytuacji, gdybyśmy chcieli umieścić jakieś obiekty w kontenerze jeszcze przed zwróceniem go do wywołującego, jak tu:

```
//: generics/FilledListMaker.java
import java.util.*;
```

```
public class FilledListMaker<T> {
  List<T> create(T t, int n) {
    List<T> result = new ArrayList<T>();
    for(int i = 0; i < n; i++)
      result.add(t);
    return result;
  }
  public static void main(String[] args) {
    FilledListMaker<String> stringMaker =
      new FilledListMaker<String>();
    List<String> list = stringMaker.create("Ahoj", 4);
    System.out.println(list);
  }
} /* Output:
[Ahoj, Ahoj, Ahoj, Ahoj]
*///:~
```

Choć kompilator nie wie nic o typie T wewnątrz metody `create()`, wciąż może — w fazie kompilacji — zapewnić, żeby to, co umieszcza się w zmiennej `result`, było typu T, tak aby zachować zgodność z `ArrayList<T>`. Mimo zatarcia informacji o faktycznym typie wewnątrz metody czy klasy kompilator wciąż może zadbać o wewnętrzną spójność użycia typu w obrębie tej metody czy klasy.

Ponieważ zacieranie usuwa informacje o typie z ciała metody, w czasie wykonania liczą się już jedynie *granice* (ang. *boundaries*) typów: miejsce, w których obiekt wkracza do metody, i miejsce, w którym go opuszcza. W tych właśnie miejscach kompilator w czasie kompilacji realizuje kontrolę typów z użyciem rzutowania. Weźmy za przykład poniższy kod:

```
//: generics/SimpleHolder.java

public class SimpleHolder {
  private Object obj;
  public void set(Object obj) { this.obj = obj; }
  public Object get() { return obj; }
  public static void main(String[] args) {
    SimpleHolder holder = new SimpleHolder();
    holder.set("Element");
    String s = (String)holder.get();
  }
} ///:~
```

Dekompilacja powyższego programu poleceniem `javap -c SimpleHolder` da w wyniku coś takiego (poniższy listing został poddany wstępnej obróbce w celu zwiększenia czytelności):

```
public void set(java.lang.Object);
  Code:
   0:aload_0
   1:aload_1
   2:putfield #2; //Field obj:Object;
   5:return

public java.lang.Object get();
  Code:
   0:aload_0
   1:getfield #2; //Field obj:Object;
   4:areturn
```

```
public static void main(java.lang.String[]);
  Code:
   0:new      #3; //class SimpleHolder
   3:dup
   4:invokespecial     #4; //Method "<init>":()V
   7:astore_1
   8:aload_1
   9:ldc      #5; //String Element
   11:        invokevirtual     #6; //Method set:(Object;)V
   14:        aload_1
   15:        invokevirtual     #7; //Method get:()Object;
   18:        checkcast         #8; //class java/lang/String
   21:        astore_2
   22:        return
```

Metody `set()` i `get()` sprowadzają się do zachowywania i zwracania wartości, a kontrola rzutowania (ang. *checkcast*) odbywa się w miejscu wywołania `get()`.

Zobaczmy teraz, jak zachowa się kod uzupełniony o uogólnienia:

```
//: generics/GenericHolder.java

public class GenericHolder<T> {
  private T obj;
  public void set(T obj) { this.obj = obj; }
  public T get() { return obj; }
  public static void main(String[] args) {
    GenericHolder<String> holder =
      new GenericHolder<String>();
    holder.set("Element");
    String s = holder.get();
  }
} ///:~
```

Nie ma już potrzeby rzutowania wyniku metody `get()`, wiemy za to, że typ wartości przekazywanej do metody `set()` jest kontrolowany w czasie kompilacji. Oto kod bajtowy takiego programu:

```
public void set(java.lang.Object);
  Code:
   0:aload_0
   1:aload_1
   2:putfield #2; //Field obj:Object;
   5:return

public java.lang.Object get();
  Code:
   0:aload_0
   1:getfield #2; //Field obj:Object;
   4:areturn

public static void main(java.lang.String[]);
  Code:
   0:new      #3; //class GenericHolder
   3:dup
   4:invokespecial     #4; //Method "<init>":()V
   7:astore_1
```

```
8: aload_1
9: ldc      #5; //String Element
11:         invokevirtual    #6; //Method set:(Object;)V
14:         aload_1
15:         invokevirtual    #7; //Method get:()Object;
18:         checkcast        #8; //class java/lang/String
21:         astore_2
22:         return
```

Okazuje się, że kod wynikowy obu programów jest identyczny. Dodatkowe operacje związane ze statyczną kontrolą typów w metodzie `set()` odbywają się na etapie kompilacji i nie ma po nich śladu w kodzie bajtowym, a rzutowanie wartości zwracanej z `get()` odbywa się i tak, ale przecież tak czy owak musielibyśmy je wykonać — fakt, że dba o to kompilator, zwiększa czytelność i zwięzłość kodu.

Ponieważ metody `get()` i `set()` dają w obu przykładach identyczne kody bajtowe, całość obsługi uogólnień — dodatkowa kontrola typów w czasie kompilacji dla wartości przekazywanych i rzutowanie wartości zwracanych — odbywa się na granicach. Ewentualne pomieszanie wynikające z techniki zacierania można zmniejszyć, pamiętając, że „granica wyznacza miejsce wykonywania czynności".

Kompensacja zacierania

Przekonaliśmy się, że zacieranie uniemożliwia wykonywanie niektórych operacji w kodzie uogólnionym. Mianowicie chodzi o te operacje, które w czasie wykonania wymagają informacji o dokładnym typie:

```
//: generics/Erased.java
// {CompileTimeError} (Nie skompiluje się)

public class Erased<T> {
  private final int SIZE = 100;
  public static void f(Object arg) {
    if(arg instanceof T) {}          // Błąd
    T var = new T();                 // Błąd
    T[] array = new T[SIZE];         // Błąd
    T[] array = (T[])new Object[SIZE]; // Ostrzeżenie niesprawdzane
  }
} ///:~
```

Niekiedy można te ograniczenia wyminąć, ale czasem trzeba skompensować niekorzystne efekty zacierania, uciekając się do stosowania *znacznika typu* (ang. *type tag*). Polega to na jawnym przekazaniu obiektu `Class` reprezentującego dany typ i wykorzystywanie tego obiektu w wyrażeniach wymagających informacji o typie.

W ramach przykładu wrócimy do poprzedniego listingu, gdzie wywołanie operatora `instanceof` było nieskuteczne z powodu zatarcia informacji o typie. Po wprowadzeniu znacznika typu można to wyrażenie zastąpić dynamicznym wywołaniem `isInstance()`:

```
//: generics/ClassTypeCapture.java

class Building {}
class House extends Building {}
```

```
public class ClassTypeCapture<T> {
  Class<T> kind;
  public ClassTypeCapture(Class<T> kind) {
    this.kind = kind;
  }
  public boolean f(Object arg) {
    return kind.isInstance(arg);
  }
  public static void main(String[] args) {
    ClassTypeCapture<Building> ctt1 =
      new ClassTypeCapture<Building>(Building.class);
    System.out.println(ctt1.f(new Building()));
    System.out.println(ctt1.f(new House()));
    ClassTypeCapture<House> ctt2 =
      new ClassTypeCapture<House>(House.class);
    System.out.println(ctt2.f(new Building()));
    System.out.println(ctt2.f(new House()));
  }
} /* Output:
true
true
false
true
*///:~
```

Kompilator zadba o to, aby znacznik typu pasował do argumentu uogólnienia.

Ćwiczenie 21. Zmodyfikuj program *ClassTypeCapture.java* przez dodanie kontenera `Map<String, Class<?>>`, metody `addType(String typename, Class<?> kind)` i metody `createNew(String typename)`. Metoda `createNew()` ma utworzyć nowy egzemplarz klasy reprezentowanej ciągiem argumentu albo wypisać komunikat o błędzie (4).

Tworzenie egzemplarzy typów

Próba wywołania `new T()` w programie *Erased.java* nie zadziała, częściowo z uwagi na zacieranie informacji o typie, a częściowo dlatego, że kompilator nie może zweryfikować, czy T posiada konstruktor domyślny (bezargumentowy). Z kolei w języku C++ taka operacja jest zupełnie naturalna, prosta i bezpieczna (bo podlega kontroli w czasie kompilacji):

```
//: generics/InstantiateGenericType.cpp
// C++, nie Java!

template<class T> class Foo {
  T x; // Deklaracja pola typu T
  T* y; // Wskaźnik typu T
public:
  // Inicjalizacja wskaźnika:
  Foo() { y = new T(); }
};

class Bar {};

int main() {
```

```
  Foo<Bar> fb;
  Foo<int> fi; // ... działa nawet z typami podstawowymi
} ///:~
```

W języku Java trzeba w takim przypadku przekazać obiekt-wytwórnię i za jego pośrednictwem utworzyć nowy egzemplarz. Wygodną wytwórnią jest obiekt `Class` reprezentujący dany typ, co po raz kolejny uzasadnia zastosowanie znacznika typu — po to, aby można było wywołać na jego rzecz metodę `newInstance()` tworzącą obiekt tego typu:

```
//: generics/InstantiateGenericType.java
import static net.mindview.util.Print.*;

class ClassAsFactory<T> {
  T x;
  public ClassAsFactory(Class<T> kind) {
    try {
      x = kind.newInstance();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
}

class Employee {}

public class InstantiateGenericType {
  public static void main(String[] args) {
    ClassAsFactory<Employee> fe =
      new ClassAsFactory<Employee>(Employee.class);
    print("Udało się utworzyć obiekt ClassAsFactory<Employee>");
    try {
      ClassAsFactory<Integer> fi =
        new ClassAsFactory<Integer>(Integer.class);
    } catch(Exception e) {
      print("Nie udało się utworzyć obiektu ClassAsFactory<Integer>");
    }
  }
} /* Output:
Udało się utworzyć obiekt ClassAsFactory<Employee>
Nie udało się utworzyć obiektu ClassAsFactory<Integer>
*///:~
```

Taki kod da się skompilować, ale próba utworzenia egzemplarza `CreateAsFactory<Integer>` zawiedzie, bo klasa `Integer` nie posiada konstruktora domyślnego. Ponieważ błąd nie jest wychwytywany w czasie kompilacji, takie podejście jest odradzane przez programistów firmy Sun. Sugerują oni, aby wykorzystywać jawną wytwórnię z ograniczeniem typu do klas implementujących interfejs wytwórni:

```
//: generics/FactoryConstraint.java

interface FactoryI<T> {
  T create();
}

class Foo2<T> {
  private T x;
  public <F extends FactoryI<T>> Foo2(F factory) {
```

```
    x = factory.create();
  }
  // ...
}

class IntegerFactory implements FactoryI<Integer> {
  public Integer create() {
    return new Integer(0);
  }
}

class Widget {
  public static class Factory implements FactoryI<Widget> {
    public Widget create() {
      return new Widget();
    }
  }
}

public class FactoryConstraint {
  public static void main(String[] args) {
    new Foo2<Integer>(new IntegerFactory());
    new Foo2<Widget>(new Widget.Factory());
  }
} ///:~
```

W istocie mamy tu do czynienia z wariacją na temat przekazywania `Class<T>`. W obu podejściach przekazywany jest obiekt wytwórni: `Class<T>` jest wytwórnią wbudowaną, podczas gdy w podejściu reprezentowanym powyżej tworzymy jawne obiekty-wytwórnie. Całość podlega kontroli w czasie kompilacji.

Inne rozwiązanie polega na implementacji wzorca projektowego *Template Method* (metoda szablonowa). W poniższym przykładzie zadanie metody szablonowej pełni metoda `get()`; metoda `create()`, zdefiniowana w podklasie, tworzy obiekt tego typu:

```
//: generics/CreatorGeneric.java

abstract class GenericWithCreate<T> {
  final T element;
  GenericWithCreate() { element = create(); }
  abstract T create();
}

class X {}

class Creator extends GenericWithCreate<X> {
  X create() { return new X(); }
  void f() {
    System.out.println(element.getClass().getSimpleName());
  }
}

public class CreatorGeneric {
  public static void main(String[] args) {
    Creator c = new Creator();
    c.f();
  }
```

```
} /* Output:
X
*///:~
```

Ćwiczenie 22. Za pomocą znacznika typu i mechanizmu refleksji utwórz metodę, która za pośrednictwem wywołania `newInstance()` w wersji z argumentami utworzy obiekt pewnej klasy, która to klasa udostępnia konstruktor z argumentami wywołania (a nie konstruktor bezargumentowy) (6).

Ćwiczenie 23. Zmodyfikuj przykład *FactoryConstraint.java* tak, aby metoda `create()` przyjmowała argument (1).

Ćwiczenie 24. Zmodyfikuj ćwiczenie 21. tak, aby obiekty wytwórni były przechowywane w kontenerze `Map` (3).

Tablice typów ogólnych

W programie *Erased.java* ujawniła się niemożność tworzenia tablic elementów typów uogólnionych. Ogólne rozwiązanie tego problemu polega na zastępowaniu zwykłych tablic kontenerami `ArrayList`:

```
//: generics/ListOfGenerics.java
import java.util.*;

public class ListOfGenerics<T> {
  private List<T> array = new ArrayList<T>();
  public void add(T item) { array.add(item); }
  public T get(int index) { return array.get(index); }
} ///:~
```

Zyskujemy zachowanie charakterystyczne dla tablicy ze statyczną kontrolą typów cechującą uogólnienia.

Mimo wszystko niekiedy pojawia się potrzeba utworzenia tablicy elementów uogólnionych (choćby we własnych implementacjach kontenerów — wszak `ArrayList` wykorzystuje wewnętrznie tablicę). Co ciekawe, oczekiwania kompilatora można zaspokoić, definiując *referencję*. Oto przykład:

```
//: generics/ArrayOfGenericReference.java

class Generic<T> {}

public class ArrayOfGenericReference {
  static Generic<Integer>[] gia;
} ///:~
```

Kompilator akceptuje taki kod bez słowa ostrzeżenia, ale nie da się utworzyć tablicy elementów dokładnego typu (z parametrami typowymi włącznie), więc jest to nieco mylące. Ponieważ wszystkie tablice mają taką samą strukturę (w zakresie rozmiaru poszczególnych pozycji i ich rozmieszczenia w pamięci), można by spróbować utworzyć tablicę elementów typu `Object`, a potem rzutować elementy na pożądany typ. Da się takie coś skompilować, ale już nie uruchomić — próba wykonania powoduje zgłoszenie wyjątku `ClassCastException`.

```
//: generics/ArrayOfGeneric.java

public class ArrayOfGeneric {
  static final int SIZE = 100;
  static Generic<Integer>[] gia;
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    // Kompiluje się, ale powoduje wyjątek ClassCastException:
    //! gia = (Generic<Integer>[])new Object[SIZE];
    // Typ czasu wykonania to typ "goły" (zatarty):
    gia = (Generic<Integer>[])new Generic[SIZE];
    System.out.println(gia.getClass().getSimpleName());
    gia[0] = new Generic<Integer>();
    //! gia[1] = new Object(); // Błąd kompilacji
    // Wykrycie niedopasowania typów w czasie kompilacji:
    //! gia[2] = new Generic<Double>();
  }
} /* Output:
Generic[]
*///:~
```

Sęk w tym, że tablica utrzymuje informacje o swoim typie, a typ ten jest ustalany w momencie utworzenia tablicy, więc mimo rzutowania `gia` na typ `Generic<Integer>[]` informacja o nowym typie istnieje jedynie dla kompilatora (a bez adnotacji `@SuppressWarnings` rzutowanie zaowocowałoby przy kompilacji komunikatem ostrzeżenia). W czasie wykonania tablica jest wciąż tablicą elementów typu `Object` i właśnie to jest przyczyną problemów. Jedynym skutecznym sposobem utworzenia tablicy elementów typu uogólnionego jest utworzenie nowej tablicy typu zatartego (a potem rzutowanie).

Przejdźmy do nieco bardziej skomplikowanego przykładu w postaci prostej uogólnionej „otoczki" tablicy:

```
//: generics/GenericArray.java

public class GenericArray<T> {
  private T[] array;
  @SuppressWarnings("unchecked")
  public GenericArray(int sz) {
    array = (T[])new Object[sz];
  }
  public void put(int index, T item) {
    array[index] = item;
  }
  public T get(int index) { return array[index]; }
  // Metoda eksponująca implementację wewnętrzną:
  public T[] rep() { return array; }
  public static void main(String[] args) {
    GenericArray<Integer> gai =
      new GenericArray<Integer>(10);
    // Powoduje wyjątek ClassCastException:
    //! Integer[] ia = gai.rep();
    // A to jest w porządku:
    Object[] oa = gai.rep();
  }
} ///:~
```

Jak poprzednio, nic możemy zapisać po prostu `T[] array = new T[sz]`, tworzymy więc tablicę elementów typu `Object` i próbujemy rzutowania.

Metoda `rep()` zwraca `T[]`, co w metodzie `main()` powinno odpowiadać tablicy `Integer[]`, ale próba wywołania metody `rep()` i przypisania wyniku do referencji `Integer[]` powoduje wyjątek `ClassCastException` — znów dlatego, że dynamiczny typ tablicy to `Object[]`.

Po oznaczeniu jako komentarz adnotacji `@SuppressWarnings` próba skompilowania programu *GenericArray.java* spowoduje wyprowadzenie komunikatu ostrzeżenia:

```
Note: GenericArray.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details
```

Otrzymamy pojedyncze ostrzeżenie, podejrzewając, że dotyczy rzutowania. Aby się o tym upewnić, należy ponownie skompilować program, tym razem z opcją `-Xlint:unchecked`:

```
GenericArray.java:7: warning: [unchecked] unchecked cast
found   : java.lang.Object[]
required: T[]
    array = (T[])new Object[sz];
                 ^

1 warning
```

Faktycznie ostrzeżenie dotyczyło rzutowania. Ponieważ komunikaty ostrzeżeń są dość rozwlekłe, najlepiej — po sprawdzeniu, czego dotyczyły — stłumić je adnotacją `@SuppressWarnings`. W ten sposób zyskujemy płynny przebieg kompilacji, zachowując czyste sumienie co do własnej staranności.

Z racji zacierania typów tablica w czasie wykonania może być tylko typu `Object[]`. Jeśli spróbujemy rzutowania wprost na typ `T[]`, w czasie kompilacji faktyczny typ tablicy zostanie utracony, przez co kompilator może stracić szansę wykonania dodatkowych operacji kontrolnych. Z tego powodu we wnętrzu własnych implementacji kolekcji najlepiej stosować tablicę `Object[]`, a rzutowanie na typ `T` wykonywać przy okazji dodawania elementu tablicy. W przykładzie *GenericArray.java* wyglądałoby to tak:

```
//: generics/GenericArray2.java

public class GenericArray2<T> {
  private Object[] array;
  public GenericArray2(int sz) {
    array = new Object[sz];
  }
  public void put(int index, T item) {
    array[index] = item;
  }
  @SuppressWarnings("unchecked")
  public T get(int index) { return (T)array[index]; }
  @SuppressWarnings("unchecked")
  public T[] rep() {
    return (T[])array; // Uwaga: niesprawdzane rzutowanie
  }
  public static void main(String[] args) {
    GenericArray2<Integer> gai =
      new GenericArray2<Integer>(10);
```

```
    for(int i = 0; i < 10; i ++)
      gai.put(i, i);
    for(int i = 0; i < 10; i ++)
      System.out.print(gai.get(i) + " ");
    System.out.println();
    try {
      Integer[] ia = gai.rep();
    } catch(Exception e) { System.out.println(e); }
  }
} /* Output: (Sample)
0 1 2 3 4 5 6 7 8 9
java.lang.ClassCastException: [Ljava.lang.Object; cannot be cast to [Ljava.lang.Integer;
*///:~
```

Na pierwszy rzut oka różnice są niewielkie, bo sprowadzają się w zasadzie do przesunięcia momentu rzutowania. Bez adnotacji `@SuppressWarnings` kompilacji wciąż towarzyszą ostrzeżenia. Ale tym razem wewnętrzna tablica to nie `T[]`, a `Object[]`. Przy wywołaniu metody `get()` następuje rzutowanie obiektu przechowywanego w tablicy na typ `T`, który jest pożądanym typem docelowym i faktycznym typem przechowywanych elementów, więc rzutowanie jest bezpieczne. Jednakże w metodzie `rep()` znów następuje próba rzutowania typu tablicy z `Object[]` na `T[]`, co wciąż jest tak samo niepoprawne, jak wcześniej i w czasie kompilacji powoduje wystosowanie ostrzeżenia, a w czasie wykonania — zgłoszenie wyjątku. Najwyraźniej nie ma sposobu określenia typu tablicy, która musi być typu `Object[]`. Zaleta traktowania wewnętrznej tablicy `array` jako typu `Object[]` zamiast `T[]` polega na zmniejszeniu prawdopodobieństwa zapomnienia o faktycznym typie tablicy i przypadkowego wprowadzenia błędu (choć błędy akurat tego rodzaju dają się szybko wykrywać w czasie wykonania).

W nowo tworzonym kodzie należałoby korzystać szeroko ze znaczników typu. Klasa `GenericArray` wykorzystująca tę technikę wyglądałaby tak:

```
//: generics/GenericArrayWithTypeToken.java
import java.lang.reflect.*;

public class GenericArrayWithTypeToken<T> {
  private T[] array;
  @SuppressWarnings("unchecked")
  public GenericArrayWithTypeToken(Class<T> type, int sz) {
    array = (T[])Array.newInstance(type, sz);
  }
  public void put(int index, T item) {
    array[index] = item;
  }
  public T get(int index) { return array[index]; }
  // Eksponowanie reprezentacji wewnętrznej:
  public T[] rep() { return array; }
  public static void main(String[] args) {
    GenericArrayWithTypeToken<Integer> gai =
      new GenericArrayWithTypeToken<Integer>(
        Integer.class, 10);
    // Teraz zadziała:
    Integer[] ia = gai.rep();
  }
} ///:~
```

Do konstruktora przekazywany jest znacznik typu `Class<T>`, który ma umożliwić zachowanie zatartych informacji, aby dało się tworzyć tablicę właściwego typu, mimo że wciąż trzeba tłumić ostrzeżenia o niebezpiecznym rzutowaniu adnotacją `@SuppressWarnings`. Dysponując pożądanym typem, możemy zwrócić obiekt tego typu i uzyskać pożądane rezultaty (co widać w metodzie `main()`). Typem tablicy w czasie wykonania jest tu `T[]`.

Niestety, gdybyś przyjrzał się kodowi źródłowemu bibliotek standardowych Java SE5, znalazłbyś tam niezliczone rzutowania tablic elementów typu `Object` na typy sparametryzowane. Przykładem może być choćby konstruktor kopiujący `ArrayList` tworzący kontener na podstawie kolekcji (kod został oczyszczony ze zbędnych elementów):

```
public ArrayList(Collection c) {
  size = c.size();
  elementData = (E[]) new Object[size];
  c.toArray(elementData);
}
```

Jeśli zajrzysz do pliku *ArrayList.java*, znajdziesz całe mnóstwo takich operacji rzutowania. A co się dzieje przy ich kompilacji?

```
Note: ArrayList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details
```

Niewątpliwie kompilacja bibliotek standardowych owocuje licznymi ostrzeżeniami. Każdy, kto programował w C, zwłaszcza w wydaniach poprzedzających ANSI C, pamięta efekt natłoku ostrzeżeń: kiedy raz się odkryje, że można je zignorować bez większych konsekwencji, przestają mieć jakiekolwiek znaczenie ostrzegawcze. Z tego powodu najlepiej byłoby powstrzymywać kompilator od wypisywania ostrzeżeń, jeśli i tak programista nie będzie miał co z nimi zrobić.

Neal Gafter (jeden z głównych programistów Javy SE5) na swojej stronie WWW[3] przyznaje, że nie pałał ochotą do przepisywania bibliotek standardowych Javy i że nie powinien był robić tego, co zrobił (odnośnie rzutowania typu tablicy w bibliotekach standardowych). Neil zaznaczał, że nie mógł poprawić części kodu bibliotecznego bez naruszania istniejącego interfejsu. Więc mimo że w kodzie biblioteki standardowej można obserwować pewne idiomy, nie należy ich od razu brać za wzorzec. Szkoda, bo kod źródłowy biblioteki standardowej języka powinien być wzorem do naśladowania dla programistów-adeptów języka.

Ramy

Pojęcie *ramy* wprowadziłem już wcześniej. Otóż ramy pozwalają na nałożenie ograniczeń na typy konkretyzujące uogólnienie. Pozwala to na narzucenie reguł co do typów dających się stosować z danym uogólnieniem; ważniejsze jest jednak, że nałożenie ramy pozwala na wywoływanie w kodzie uogólnienia metod właściwych dla typów mieszczących się w tych ramach.

[3] *http://gafter.blogspot.com/2004/09/puzzling-through-erasure-answer.html*

Ponieważ zacieranie usuwa informacje o typach, jedynymi metodami, które można wywoływać z wnętrza kodu uogólnionego pozbawionego ram konkretyzacji, są metody dostępne w klasie Object. Gdyby jednak ograniczyć zakres potencjalnych typów konkretyzacji uogólnienia, można by wywoływać metody należące do typów z tego zakresu. Ograniczenie takie jest w języku Java sygnalizowane słowem extends. Wypada zaznaczyć, że słowo kluczowe extends ma w kontekście uogólnień zupełnie inne znaczenie niż normalnie, to znaczy w nagłówku definicji klasy. Podstawy stosowania ram konkretyzacji ilustruje poniższy przykład:

```
//: generics/BasicBounds.java

interface HasColor { java.awt.Color getColor(); }

class Colored<T extends HasColor> {
  T item;
  Colored(T item) { this.item = item; }
  T getItem() { return item; }
  // Obramowanie typów konkretyzujących pozwala na wywołanie metody:
  java.awt.Color color() { return item.getColor(); }
}

class Dimension { public int x, y, z; }

// To nie zadziała -- najpierw klasa, potem interfejsy:
// class ColoredDimension<T extends HasColor & Dimension> {

// Ramy wielokrotne:
class ColoredDimension<T extends Dimension & HasColor> {
  T item;
  ColoredDimension(T item) { this.item = item; }
  T getItem() { return item; }
  java.awt.Color color() { return item.getColor(); }
  int getX() { return item.x; }
  int getY() { return item.y; }
  int getZ() { return item.z; }
}

interface Weight { int weight(); }

// Tak jak przy dziedziczeniu można mieć tylko jedną
// klasę konkretną, za to z wieloma interfejsami:
class Solid<T extends Dimension & HasColor & Weight> {
  T item;
  Solid(T item) { this.item = item; }
  T getItem() { return item; }
  java.awt.Color color() { return item.getColor(); }
  int getX() { return item.x; }
  int getY() { return item.y; }
  int getZ() { return item.z; }
  int weight() { return item.weight(); }
}

class Bounded
extends Dimension implements HasColor, Weight {
  public java.awt.Color getColor() { return null; }
  public int weight() { return 0; }
}
```

```
public class BasicBounds {
  public static void main(String[] args) {
    Solid<Bounded> solid =
      new Solid<Bounded>(new Bounded());
    solid.color();
    solid.getY();
    solid.weight();
  }
} ///:~
```

Łatwo zauważyć, że program *BasicBounds.java* już na pierwszy rzut oka zawiera elementy nadmiarowości, które można by wyeliminować za pomocą dziedziczenia. Zobaczmy, jak kolejne poziomy dziedziczenia dodają ograniczenia do ramy:

```
//: generics/InheritBounds.java

class HoldItem<T> {
  T item;
  HoldItem(T item) { this.item = item; }
  T getItem() { return item; }
}

class Colored2<T extends HasColor> extends HoldItem<T> {
  Colored2(T item) { super(item); }
  java.awt.Color color() { return item.getColor(); }
}

class ColoredDimension2<T extends Dimension & HasColor>
extends Colored2<T> {
  ColoredDimension2(T item) {  super(item); }
  int getX() { return item.x; }
  int getY() { return item.y; }
  int getZ() { return item.z; }
}

class Solid2<T extends Dimension & HasColor & Weight>
extends ColoredDimension2<T> {
  Solid2(T item) {  super(item); }
  int weight() { return item.weight(); }
}

public class InheritBounds {
  public static void main(String[] args) {
    Solid2<Bounded> solid2 =
      new Solid2<Bounded>(new Bounded());
    solid2.color();
    solid2.getY();
    solid2.weight();
  }
} ///:~
```

Klasa `HoldItem` służy do przechowywania obiektu, która to jej cecha jest dziedziczona przez klasę `Colored2`, a ta przy okazji wymaga, aby jej parametr spełniał interfejs `HasColor`. Klasy `ColoredDimension2` i `Solid2` kontynuują rozwijanie hierarchii i dodawanie ram. Teraz metody są dziedziczone i nie trzeba powtarzać ich definicji w każdej klasie z osobna.

Oto przykład z jeszcze większą liczbą poziomów:

```
//: generics/EpicBattle.java
// Ramy uogólnień w Javie.
import java.util.*;

interface SuperPower {}
interface XRayVision extends SuperPower {
  void seeThroughWalls();
}
interface SuperHearing extends SuperPower {
  void hearSubtleNoises();
}
interface SuperSmell extends SuperPower {
  void trackBySmell();
}

class SuperHero<POWER extends SuperPower> {
  POWER power;
  SuperHero(POWER power) { this.power = power; }
  POWER getPower() { return power; }
}

class SuperSleuth<POWER extends XRayVision>
extends SuperHero<POWER> {
  SuperSleuth(POWER power) { super(power); }
  void see() { power.seeThroughWalls(); }
}

class CanineHero<POWER extends SuperHearing & SuperSmell>
extends SuperHero<POWER> {
  CanineHero(POWER power) { super(power); }
  void hear() { power.hearSubtleNoises(); }
  void smell() { power.trackBySmell(); }
}

class SuperHearSmell implements SuperHearing, SuperSmell {
  public void hearSubtleNoises() {}
  public void trackBySmell() {}
}

class DogBoy extends CanineHero<SuperHearSmell> {
  DogBoy() { super(new SuperHearSmell()); }
}

public class EpicBattle {
  // Ramy w metodach uogólnionych:
  static <POWER extends SuperHearing>
  void useSuperHearing(SuperHero<POWER> hero) {
    hero.getPower().hearSubtleNoises();
  }
  static <POWER extends SuperHearing & SuperSmell>
  void superFind(SuperHero<POWER> hero) {
    hero.getPower().hearSubtleNoises();
    hero.getPower().trackBySmell();
  }
  public static void main(String[] args) {
    DogBoy dogBoy = new DogBoy();
```

```
    useSuperHearing(dogBoy);
    superFind(dogBoy);
    // Można zrobić tak:
    List<? extends SuperHearing> audioBoys;
    // Ale nie można tak:
    // List<? extends SuperHearing & SuperSmell> dogBoys;
  }
} ///:~
```

Zwróć uwagę na symbole wieloznaczne (o których za chwilę) i ich ograniczenie do pojedynczej ramy.

Ćwiczenie 25. Utwórz dwa interfejsy i klasę, która je (oba) implementuje. Utwórz dwie metody uogólnione; parametr typowy jednej z nich ma być ograniczony ramą pierwszego interfejsu, a drugiej — ramą drugiego interfejsu. Utwórz egzemplarz klasy implementującej oba interfejsy i pokaż, że można go użyć w wywołaniach obu metod uogólnionych (2).

Symbole wieloznaczne

Już w kilku przykładach pojawiły się *symbole wieloznaczne* (ang. *wildcard*) — w postaci znaków zapytania w wyrażeniach argumentów konkretyzacji uogólnienia — widniały choćby w rozdziałach „Kolekcje obiektów" i „Informacje o typach". Najwyższa pora przyjrzeć im się dokładniej.

Na początek przykład pokazujący pewien aspekt zachowania tablic: otóż można przypisać tablicę elementów typu pochodnego do referencji tablicy elementów typu bazowego:

```
// generics/CovariantArrays.java

class Fruit {}
class Apple extends Fruit {}
class Jonathan extends Apple {}
class Orange extends Fruit {}

public class CovariantArrays {
  public static void main(String[] args) {
    Fruit[] fruit = new Apple[10];
    fruit[0] = new Apple(); // OK
    fruit[1] = new Jonathan(); // OK
    // Typ w czasie wykonania to Apple[], nie Fruit[] ani Orange[]:
    try {
      // Kompilator pozwoli na dodanie egzemplarza Fruit:
      fruit[0] = new Fruit(); // Wyjątek ArrayStoreException
    } catch(Exception e) { System.out.println(e); }
    try {
      // Kompilator pozwoli też na dodanie egzemplarzy Orange:
      fruit[0] = new Orange(); // Wyjątek ArrayStoreException
    } catch(Exception e) { System.out.println(e); }
  }
} /* Output:
java.lang.ArrayStoreException: Fruit
java.lang.ArrayStoreException: Orange
*///:~
```

Pierwszy wiersz metody `main()` tworzy tablicę elementów typu `Apple` i przypisuje ją do referencji tablicy elementów typu `Fruit`. Ma to sens: `Apple` to podtyp `Fruit`, więc tablica elementów `Apple` jest równocześnie tablicą elementów `Fruit`.

Skoro jednak faktycznym typem tablicy jest `Apple[]`, w tablicy można umieszczać jedynie obiekty typu `Apple` i jego podtypów, a ograniczenie to powinno obowiązywać zarówno w czasie kompilacji, jak i w czasie wykonania. Mimo to kompilator pozwala na umieszczenie w tablicy elementu typu `Fruit`. Dla kompilatora ma to sens, bo operuje on na referencji typu `Fruit[]` — dlaczegóż miałby odmawiać umieszczenia w takiej tablicy egzemplarza `Fruit`, bądź egzemplarzy dowolnych typów pochodnych `Fruit`, choćby i `Orange`? Dlatego też próby takie przechodzą kompilację. Dopiero mechanizm obsługi tablic w czasie wykonania rozpoznaje tablicę jako obiekt typu `Apple[]` i uniemożliwia dołożenie do niej elementu niezgodnego typu, zgłaszając wyjątek.

Zastosowanie tu terminu „rzutowania w górę" nie byłoby do końca właściwe. Tak naprawdę odbywa się tu przypisanie jednej tablicy do innej. Cechą tablic jest to, że przechowują inne obiekty, ale ponieważ możemy wykonywać rzutowanie w górę, obiekty tablic mogą zachowywać reguły odnośnie typów przechowywanych elementów. To tak, jakby tablica była świadoma swojej zawartości, więc nie da się jej nadużyć pomiędzy kontrolami wykonywanymi w czasie kompilacji a kontrolą w czasie wykonania.

Dla tablic nie jest to wielce uciążliwe, bo w czasie wykonania od razu wiadomo, że doszło do próby wstawienia obiektu niewłaściwego typu. Ale jednym z zadań uogólnień jest przesuwanie takich odkryć z czasu wykonania do czasu kompilacji. Jaki efekt uzyskamy, zastępując tablice uogólnionymi kontenerami?

```
//: generics/NonCovariantGenerics.java
// {CompileTimeError} (Nie skompiluje się)
import java.util.*;

public class NonCovariantGenerics {
  // Błąd kompilacji -- niezgodność typów:
  List<Fruit> flist = new ArrayList<Apple>();
} ///:~
```

Na pierwszy rzut oka widać, że „nie można przypisać kontenera obiektów `Apple` do kontenera obiektów `Fruit`". Pamiętaj jednak, że uogólnienia nie dotyczą jedynie kontenerów. Tak naprawdę widać tu, że „nie można przypisać uogólnienia *angażującego* typ `Apple` do uogólnienia *angażującego* typ `Fruit`". Gdyby (jak w przypadku tablic) kompilator wiedział wystarczająco dużo o kodzie, żeby stwierdzić, że chodzi o kontenery, być może mógłby przymknąć oko na niezgodność typów. Jednakże żadna taka wiedza nie jest kompilatorowi dostępna, więc kompilator odmawia „rzutowania w górę". Tak naprawdę nie jest to jednak żadne rzutowanie — kontener `List` obiektów `Apple` nie jest w żadnej mierze kontenerem `List` obiektów `Fruit`. Kontener `List` obiektów `Apple` może przechowywać obiekty klasy `Apple` i ich klas pochodnych, a kontener `List` obiektów `Fruit` może przechowywać dowolne podtypy `Fruit`. Owszem, ta kategoria obejmuje też typ `Apple`, ale zachodzenie tej relacji pomiędzy typami nic czyni z kontenera `List<Fruit>` kontenera `List<Apple>`; równoważność `Apple` z `Fruit` nie oznacza równoważności typowej kontenerów obiektów `Apple` z kontenerami obiektów `Fruit`.

Chodzi bowiem nie o typy obiektów przechowywanych w kontenerze, a o typy samych kontenerów. Uogólnienia, w przeciwieństwie do tablic, nie podlegają wbudowanej kowariancji. Tablice są przecież definiowane w całości w samym języku programowania i jako takie podlegają wbudowanej kontroli w czasie kompilacji i w czasie wykonania; w przypadku uogólnień kompilator i system wykonawczy nie może wiedzieć niczego o zamiarach programisty i pożądanych przez niego regułach zgodności typów.

Niekiedy jednak można ustanowić relację rzutowania w górę pomiędzy dwoma kontenerami. Służą do tego właśnie symbole wieloznaczne:

```
//: generics/GenericsAndCovariance.java
import java.util.*;

public class GenericsAndCovariance {
  public static void main(String[] args) {
    // Symbole wieloznaczne umożliwiają kowariancję:
    List<? extends Fruit> flist = new ArrayList<Apple>();
    // Błąd kompilacji: nie można dodać obiektów żadnego typu:
    // flist.add(new Apple());
    // flist.add(new Fruit());
    // flist.add(new Object());
    flist.add(null); // Dozwolone, ale niepożądane
    // Wiadomo, że otrzymamy obiekt Fruit albo pochodny:
    Fruit f = flist.get(0);
  }
} ///:~
```

Typ kontenera `flist` został tu określony jako `List<? extends Fruit>`, co można odczytać jako „lista elementów dowolnego typu wyprowadzonego z `Fruit`". Nie można jednak niczego powiedzieć o samym typie przechowywanych elementów. Symbol wieloznaczny odnosi się do faktycznego typu, czyli „jakiegoś typu, którego referencja `flist` nie określa". Przypisywany do tej referencji kontener powinien przechowywać obiekty typów takich jak `Fruit` czy `Apple`, ale na potrzeby przypisania do `flist` typ tej referencji można opisać słowem „nieważne".

Jeśli jedynym ograniczeniem utworzonego kontenera jest ograniczenie typów przechowywanych elementów do hierarchii `Fruit`, choć faktyczny typ jest nieistotny, to co można zrobić z takim kontenerem? Skoro nie wiadomo, jakiego typu elementy przechowuje, czy można bezpiecznie dodać do niego obiekt? Otóż nie można, z tym że tym razem o tej niemożności przekonamy się już na etapie kompilacji, podczas gdy w przypadku tablicy komunikował ją dopiero wyjątek czasu wykonania.

Czy to nie jest pewna przesada, skoro nie można dodać do kontenera obiektu `Apple`, mimo że przypisywany kontener był deklarowany jako przechowujący obiekty właśnie typu `Apple`? Cóż, kompilator nic o tym nie wie. Dla niego referencja typu `List<? extends Fruit>` może równie dobrze odnosić się do kontenera `List<Orange>`, a skoro tak, to do kontenera nie można dołożyć żadnego obiektu, nawet klasy `Object`.

Z drugiej strony, jeśli wywołasz metodę udostępniającą zawartość kontenera, to możesz śmiało przypisać jej wynik do referencji `Fruit`, bo wiadomo, że wszystko, co znajduje się w kontenerze, musi być co najmniej typu `Fruit`, ewentualnie jednego z typów bardziej specjalizowanych (pochodnych).

Ćwiczenie 26. Zademonstruj kowariancję tablic z użyciem klas `Number` i `Integer` (2).

Ćwiczenie 27. Pokaż (znów korzystając z klas `Number` i `Integer`), że kowariancja nie dotyczy uogólnionych kontenerów, a następnie spróbuj z symbolami wieloznacznymi (2).

Jak bystry jest kompilator?

Wziąwszy pod uwagę ostatnie doświadczenia, możemy sądzić, że nie dasz również rady wywołać żadnej metody kontenera przyjmującej argumenty. Ale spójrz tu:

```
//: generics/CompilerIntelligence.java
import java.util.*;

public class CompilerIntelligence {
  public static void main(String[] args) {
    List<? extends Fruit> flist =
      Arrays.asList(new Apple());
    Apple a = (Apple)flist.get(0); // Bez ostrzeżenia
    flist.contains(new Apple()); // Argument jest typu 'Object'
    flist.indexOf(new Apple()); // Argument jest typu 'Object'
  }
} ///:~
```

Metody `contains()` i `indexOf()` z powodzeniem przyjmują argumenty w postaci obiektów klasy `Apple`. Czy to oznacza, że kompilator faktycznie przeanalizował kod źródłowy i stwierdził, że wywoływana metoda nie modyfikuje zawartości kontenera i można ją zrealizować?

Przejrzenie dokumentacji klasy `ArrayList` pozwoli stwierdzić, że kompilator nie jest aż tak bystry. Otóż metoda `add()` kontenera przyjmuje argument typu uogólnionego, ale już `contains()` i `indexOf()` są deklarowane jako przyjmujące argumenty typu `Object`. Po zadeklarowaniu kontenera jako typu `ArrayList<? extends Fruit>` typ argument wywołania `add()` ma typ `<? extends Fruit>`. Taki opis typu nie pozwala kompilatorowi stwierdzić, o który z podtypów `Fruit` chodzi, więc profilaktycznie nie przyjmie żadnego z tych typów. Nawet uprzednie rzutowanie argumentu w górę, na typ `Fruit`, nic zmieni sytuacji — kompilator po prostu odmówi wywołania metody (tu `add()`), która w określeniu typu argumentów posiada symbole wieloznaczne.

W metodach `contains()` i `indexOf()` argumenty są deklarowane jako obiekty typu `Object`, a więc bez żadnych symboli wieloznacznych — dzięki temu kompilator może z czystym sumieniem zrealizować wywołania tych metod nawet na rzecz kontenera, którego typ angażuje symbole wieloznaczne. Jak widać, to projektant klasy uogólnionej decyduje, które z operacji są bezpieczne, sygnalizując to argumentami typu `Object`. Z kolei wywołania metod wrażliwych na typ można zablokować argumentami, których typy odwołują się do parametrów typowych uogólnienia.

Można to sprawdzić na bardzo prostej klasie `Holder`:

```
//: generics/Holder.java

public class Holder<T> {
  private T value;
```

```
  public Holder() {}
  public Holder(T val) { value = val; }
  public void set(T val) { value = val; }
  public T get() { return value; }
  public boolean equals(Object obj) {
    return value.equals(obj);
  }
  public static void main(String[] args) {
    Holder<Apple> Apple = new Holder<Apple>(new Apple());
    Apple d = Apple.get();
    Apple.set(d);
    // Holder<Fruit> Fruit = Apple; // Nie można rzutować w górę
    Holder<? extends Fruit> fruit = Apple; // OK
    Fruit p = fruit.get();
    d = (Apple)fruit.get(); // Zwraca 'Object'
    try {
      Orange c = (Orange)fruit.get(); // Bez ostrzeżenia
    } catch(Exception e) { System.out.println(e); }
    // fruit.set(new Apple()); // Nie można wywołać metody set()
    // fruit.set(new Fruit()); // Nie można wywołać metody set()
    System.out.println(fruit.equals(d)); // OK
  }
} /* Output: (Sample)
java.lang.ClassCastException: Apple cannot be cast to Orange
true
*///:~
```

Klasa `Holder` posiada metodę `set()` przyjmującą argument typu `T`, metodę `get()` zwracającą wartość typu `T` i metodę `equals()` przyjmującą argument typu `Object`. Wiemy już, że po utworzeniu egzemplarza `Holder<Apple>` nie możemy rzutować go w górę na typ `Holder<Fruit>`, można za to dokonać „rzutowania" na `Holder<? extends Fruit>`. Wywołanie `get()` dla tak określonego kontenera zwróci `Fruit` — bo tylko do tego stopnia może sprecyzować typ wartości zwracanej, przy ramie konkretyzacji określonej jako „cokolwiek, co dziedziczy po `Fruit`". Jeśli programista ma lepsze informacje na temat faktycznego typu wartości zwracanej, może samodzielnie wykonać rzutowanie na konkretny podtyp `Fruit`, i to bez ostrzeżenia, ryzykując za to wyjątek `ClassCastException`. Metoda `set()` nie zadziała ani z obiektem klasy `Apple`, ani z egzemplarzem `Fruit`, bo argument typu `set()` został określony jako `<? extends Fruit>`, a więc typ bliżej nieokreślony, z którego weryfikacją kompilator nie może sobie nijak poradzić.

Ale już metoda `equals()` radzi sobie lepiej, bo przyjmuje argument nie typu `T`, a konkretnego typu `Object`. Jak widać, kompilator kontroluje jedynie typy zwracane i przekazywane. Nie analizuje kodu w celu określenia charakteru wykonywanych operacji i ewentualnej możliwości rezygnacji z kontroli typów.

Kontrawariancja

Można też pójść inną drogą i skorzystać z *symbolu wieloznacznego nadtypu*. Taki symbol wieloznaczny wyznacza granicę ramy konkretyzacji jako dowolną klasę bazową klasy podanej za słowem `super`: można więc zapisać `<? super Klasa>` albo nawet `<? super T>` (nie można z kolei określić ramy dla typu ogólnego jako nadklasy, czyli `<T super Klasa>`).

Takie ograniczenie konkretyzacji uogólnienia pozwala na bezpieczne przekazywanie do typu uogólnionego obiektu typowanego. Symbole wieloznaczne nadtypu umożliwiają na przykład umieszczanie obiektów w uogólnionej kolekcji:

```
//: generics/SuperTypeWildcards.java
import java.util.*;

public class SuperTypeWildcards {
  static void writeTo(List<? super Apple> apples) {
    apples.add(new Apple());
    apples.add(new Jonathan());
    // apples.add(new Fruit()); // Błąd
  }
} ///:~
```

Argument `apples` to kontener `List` elementów typu bazowego względem `Apple`; wiadomo, że można do takiego kontenera całkiem bezpiecznie dodawać obiekty typu `Apple` i typów pochodnych. Ponieważ jednak *dolna krawędź ramy* konkretyzacji to `Apple`, nie sposób odpowiedzialnie stwierdzić bezpieczeństwa dodania do kontenera obiektu typu `Fruit`, bo to otwierałoby kontener na dodawanie typów niebędących `Apple`, co zniweczyłoby statyczne bezpieczeństwo typów.

Ramy konkretyzacji w postaci nadtypów i podtypów można więc rozpatrywać jako warunki bezpiecznego „zapisu" (przekazywania obiektów do metod) typu uogólnionego i jego „odczytu" (pobierania wartości zwracanych).

Ograniczenie względem typu bazowego rozluźnia ograniczenie co do tego, co można przekazywać do metody:

```
//: generics/GenericWriting.java
import java.util.*;

public class GenericWriting {
  static <T> void writeExact(List<T> list, T item) {
    list.add(item);
  }
  static List<Apple> apples = new ArrayList<Apple>();
  static List<Fruit> fruit = new ArrayList<Fruit>();
  static void f1() {
    writeExact(apples, new Apple());
    // writeExact(fruit, new Apple()); // Błąd:
    // niezgodność typów
  }
  static <T> void
  writeWithWildcard(List<? super T> list, T item) {
    list.add(item);
  }
  static void f2() {
    writeWithWildcard(apples, new Apple());
    writeWithWildcard(fruit, new Apple());
  }
  public static void main(String[] args) { f1(); f2(); }
} ///:~
```

Metoda `writeExact()` wykorzystuje jednoznaczny parametr typowy (bez symboli wieloznacznych). W metodzie `f1()` widać, że wywołania `writeExact()` działają dopóty, dopóki do listy `List<Apple>` dodajemy obiekty `Apple`. Nie można jednak za pomocą tej metody umieścić obiektu `Apple` w `List<Fruit>`.

W metodzie `writeWithWildcard()` argument wywołania ma typ `<? super T>`, więc lista `List` przechowuje obiekty pewnego typu bazowego względem `T`; zakładamy niniejszym, że do metod `List` można bezpiecznie przekazywać obiekty typu zgodnego z deklarowanym typem elementów albo typu pochodnego. Widać to w `f2()`, gdzie obiekty `Apple` możemy dodawać do kontenera `List<Apple>`, ale możemy je także dodawać do kontenera `List<Fruit>`.

Podobnej analizie można poddać symbole wieloznaczne w kontekście kowariancji:

```
//: generics/GenericReading.java
import java.util.*;

public class GenericReading {
  static <T> T readExact(List<T> list) {
    return list.get(0);
  }
  static List<Apple> apples = Arrays.asList(new Apple());
  static List<Fruit> fruit = Arrays.asList(new Fruit());
  // Statyczna metoda adaptująca wywołania:
  static void f1() {
    Apple a = readExact(apples);
    Fruit f = readExact(fruit);
    f = readExact(apples);
  }
  // Konkretny typ klasy jest ustalany w momencie
  // utworzenia egzemplarza klasy:
  static class Reader<T> {
    T readExact(List<T> list) { return list.get(0); }
  }
  static void f2() {
    Reader<Fruit> fruitReader = new Reader<Fruit>();
    Fruit f = fruitReader.readExact(fruit);
    // Fruit a = fruitReader.readExact(apples); // Błąd:
    // readExact(List<Fruit>) cannot be
    // applied to (List<Apple>).
  }
  static class CovariantReader<T> {
    T readCovariant(List<? extends T> list) {
      return list.get(0);
    }
  }
  static void f3() {
    CovariantReader<Fruit> fruitReader =
      new CovariantReader<Fruit>();
    Fruit f = fruitReader.readCovariant(fruit);
    Fruit a = fruitReader.readCovariant(apples);
  }
  public static void main(String[] args) {
    f1(); f2(); f3();
  }
} ///:~
```

Jak poprzednio, pierwsza z metod (`readExact()`) operuje na dokładnym typie. Jeśli więc użyjemy konkretnego typu bez symboli wieloznacznych, będziemy mogli i zapisywać, i odczytywać obiekty tego typu do i z kontenera. Dodatkowo odnośnie wartości zwracanej statyczna metoda uogólniona `readExact()` adaptuje każde wywołanie metody i z kontenera `List<Apple>` zwraca `Apple`, a z kontenera `List<Fruit>` zwraca `Fruit` — widać to w `f1()`. Okazuje się, że w operacji odczytu nie trzeba uciekać się do kowariancji, o ile można zastosować taką metodę statyczną.

Z kolei w przypadku uogólnienia klasy parametr typowy jest precyzowany w momencie utworzenia egzemplarza klasy. W metodzie `f2()` widać, że egzemplarz `fruitReader` może odczytywać obiekty `Fruit` z listy `List<Fruit>`, bo operuje dokładnie na zgodnym typie. Jednakże obiekty `Fruit` powinniśmy móc wydobywać również z kontenera `List<Apple>`, a tego już `fruitReader` nie potrafi.

Rozwiązaniem jest metoda `CovariantReader.readCovariant()` przyjmująca argument typu `List<? extends T>`, dzięki czemu można bezpiecznie odczytywać `T` z kontenera (wiadomo bowiem, że wszystko, co znajduje się w kontenerze, jest typu `T` albo typu pochodnego). Metoda `f3()` ilustruje zyskaną w ten sposób możliwość odczytywania obiektów `Fruit` z kontenera `List<Apple>`.

Ćwiczenie 28. Utwórz klasę uogólnioną `Generic<T>` z pojedynczą metodą przyjmującą argument typu `T`. Utwórz drugą klasę uogólnioną, `Generic2<T>`, z pojedynczą metodą zwracającą argument typu `T`. Napisz metodę uogólnioną z kontrawariancyjnym argumentem pierwszej klasy uogólnionej, która wywołuje jej metodę. Napisz drugą metodę uogólnioną z kowariancyjnym argumentem drugiej klasy uogólnionej, która wywołuje jej metodę. Przetestuj całość na bibliotece `typeinfo.pets` (4).

Symbole wieloznaczne bez ram konkretyzacji

Symbol wieloznaczny pozbawiony ramy konkretyzacji (ang. *unbounded wildcard*), zapisywany `<?>`, oznacza „cokolwiek"— wydaje się więc, że uogólnienia z takim symbolem mogłoby równie dobrze nie być. W rzeczy samej kompilator wydaje się z początku zgadzać z takim podejściem do sprawy:

```
//: generics/UnboundedWildcards1.java
import java.util.*;

public class UnboundedWildcards1 {
  static List list1;
  static List<?> list2;
  static List<? extends Object> list3;
  static void assign1(List list) {
    list1 = list;
    list2 = list;
    // list3 = list; // Ostrzeżenie: niesprawdzana konwersja
    // (Found: List, Required: List<? extends Object>)
  }
  static void assign2(List<?> list) {
    list1 = list;
    list2 = list;
    list3 = list;
  }
```

```
  static void assign3(List<? extends Object> list) {
    list1 = list;
    list2 = list;
    list3 = list;
  }
  public static void main(String[] args) {
    assign1(new ArrayList());
    assign2(new ArrayList());
    // assign3(new ArrayList()); // Ostrzeżenie:
    // Niesprawdzona konwersja (Found: ArrayList
    // Required: List<? extends Object>)
    assign1(new ArrayList<String>());
    assign2(new ArrayList<String>());
    assign3(new ArrayList<String>());
    // List<?> dopuszcza obie formy:
    List<?> wildList = new ArrayList();
    wildList = new ArrayList<String>();
    assign1(wildList);
    assign2(wildList);
    assign3(wildList);
  }
} ///:~
```

Istnieje wiele sytuacji takich jak te, w których kompilator mógłby nie rozróżniać pomiędzy typem „gołym" a uogólnieniem <?>. W tych przypadkach <?> można traktować jako dekorację; ale i w takiej roli obecność symbolu wieloznacznego ma jakąś wymowę: „napisałem ten kod pod kątem uogólnień Javy i nie pomyśl sobie, że wykorzystuję tu goły typ, ale że parametr typowy może przyjmować dowolne typy".

Drugi przykład pokaże ważne zastosowanie uogólnień z symbolami wieloznacznymi bez ramy konkretyzacji. Otóż w przypadku większej liczby parametrów typowych uogólnienia niekiedy przydaje się możliwość określenia jednego z parametrów typowych przy zachowaniu nieokreśloności innego:

```
//: generics/UnboundedWildcards2.java
import java.util.*;

public class UnboundedWildcards2 {
  static Map map1;
  static Map<?,?> map2;
  static Map<String,?> map3;
  static void assign1(Map map) { map1 = map; }
  static void assign2(Map<?,?> map) { map2 = map; }
  static void assign3(Map<String,?> map) { map3 = map; }
  public static void main(String[] args) {
    assign1(new HashMap());
    assign2(new HashMap());
    // assign3(new HashMap()); // Ostrzeżenie:
    // Niesprawdzana konwersja (Found: HashMap
    // Required: Map<String,?>)
    assign1(new HashMap<String,Integer>());
    assign2(new HashMap<String,Integer>());
    assign3(new HashMap<String,Integer>());
  }
} ///:~
```

Powtarzam, tam gdzie wszystkie parametry typowe mają postać symboli wieloznacznych, jak w `Map<?,?>`, równie dobrze można by zrezygnować z uogólnienia (i napisać po prostu `Map`). Przy okazji program *UnboundedWildcards1.java* pokazuje, że kompilator różnie traktuje typy `List<?>` i `List<? extends Object>`.

Nieco mylące jest to, że kompilator nie przejmuje się zupełnie różnicą pomiędzy `List`, a dajmy na to `List<?>`; niby racja, bo po zatarciu argument uogólnienia jest doprowadzany do krawędzi ramy, więc `List<?>` to w zasadzie tyle co `List<Object>`, a samo `List` to też w sumie tyle co `List<Object>` — ale tylko „w zasadzie"; samo `List` oznacza bowiem „goły kontener `List` zdatny do przechowywania obiektów typu i dowolnego podtypu `Object`", a `List<?>` oznacza „kontener `List` dla obiektów *pewnego typu*, chwilowo nieznanego".

Kiedy kompilator rozróżnia gołe typy od uogólnień z symbolami wieloznacznymi pozbawionymi ram konkretyzacji? Przekonamy się na przykładzie wykorzystującym zdefiniowaną wcześniej klasę `Holder<T>`. Przykład zawiera metodę przyjmującą argument typu `Holder`, ale pod różnymi postaciami: raz jako goły typ, raz jako uogólnienie z parametrem typowym, a raz jako uogólnienie z symbolem wieloznacznym:

```
// : generics/Wildcards.java
// Badanie znaczenia symboli wieloznacznych w uogólnieniach.

public class Wildcards {
  // Argument gołego typu:
  static void rawArgs(Holder holder, Object arg) {
    // holder.set(arg); // Ostrzeżenie:
    //   Niesprawdzane wywołanie set(T) jako metody
    //   gołego typu Holder
    // holder.set(new Wildcards()); // To samo

    // Tak nie można; nie znamy żadnego 'T':
    // T t = holder.get();

    // OK, ale z utratą informacji o typie:
    Object obj = holder.get();
  }
  // Jak w rawArgs(), ale z błędami zamiast ostrzeżeń:
  static void unboundedArg(Holder<?> holder, Object arg) {
    // holder.set(arg); // Błąd
    //   (set(capture of ?) in Holder<capture of ?>
    //   cannot be applied to (Object))
    // holder.set(new Wildcards()); // Taki sam błąd

    // Tak nie można; nie znamy żadnego 'T':
    // T t = holder.get();

    // OK, ale z utratą informacji o typie:
    Object obj = holder.get();
  }
  static <T> T exact1(Holder<T> holder) {
    T t = holder.get();
    return t;
  }
  static <T> T exact2(Holder<T> holder, T arg) {
    holder.set(arg);
```

```
    T t = holder.get();
    return t;
  }
  static <T>
  T wildSubtype(Holder<? extends T> holder, T arg) {
    // holder.set(arg); // Błąd
    //    (set(capture of ? extends T) in
    //    Holder<capture of ? extends T>
    //    cannot be applied to (T))
    T t = holder.get();
    return t;
  }
  static <T>
  void wildSupertype(Holder<? super T> holder, T arg) {
    holder.set(arg);
    // T t = holder.get();   // Błąd
    //    (ncompatible types: found Object, required T)

    // OK, ale z utratą informacji o typie:
    Object obj = holder.get();
  }
  public static void main(String[] args) {
    Holder raw = new Holder<Long>();
    // Albo tak:
    raw = new Holder();
    Holder<Long> qualified = new Holder<Long>();
    Holder<?> unbounded = new Holder<Long>();
    Holder<? extends Long> bounded = new Holder<Long>();
    Long lng = 1L;

    rawArgs(raw, lng);
    rawArgs(qualified, lng);
    rawArgs(unbounded, lng);
    rawArgs(bounded, lng);

    unboundedArg(raw, lng);
    unboundedArg(qualified, lng);
    unboundedArg(unbounded, lng);
    unboundedArg(bounded, lng);

    // Object r1 = exact1(raw); // Ostrzeżenia
    //    (Unchecked conversion from Holder to Holder<T>,
    //    Unchecked method invocation: exact1(Holder<T>)
    //    is applied to (Holder))
    Long r2 = exact1(qualified);
    Object r3 = exact1(unbounded); // Musi zwracać Object
    Long r4 = exact1(bounded);

    // Long r5 = exact2(raw, lng); // Ostrzeżenia
    //    (Unchecked conversion from Holder to Holder<Long>
    //    Unchecked method invocation: exact2(Holder<T>,T)
    //    is applied to (Holder,Long))
    Long r6 = exact2(qualified, lng);
    // Long r7 = exact2(unbounded, lng); // Błąd
    //    (exact2(Holder<T>,T) cannot be applied to
    //    (Holder<capture of ?>,Long))
    // Long r8 = exact2(bounded, lng); // Błąd
    //    (exact2(Holder<T>,T) cannot be applied
```

```
    //    to (Holder<capture of ? extends Long>,Long))

    // Long r9 = wildSubtype(raw, lng); // Ostrzeżenia
    //    (Unchecked conversion from Holder
    //    to Holder<? extends Long>
    //    Unchecked method invocation:
    //    wildSubtype(Holder<? extends T>,T) is
    //    applied to (Holder,Long))
    Long r10 = wildSubtype(qualified, lng);
    // OK, ale może zwrócić jedynie Object:
    Object r11 = wildSubtype(unbounded, lng);
    Long r12 = wildSubtype(bounded, lng);

    // wildSupertype(raw, lng); // Ostrzeżenia
    //    (Unchecked conversion from Holder
    //    to Holder<? super Long>
    //    Unchecked method invocation:
    //    wildSupertype(Holder<? super T>,T)
    //    is applied to (Holder,Long))
    wildSupertype(qualified, lng);
    // wildSupertype(unbounded, lng); // Błąd
    //    (wildSupertype(Holder<? super T>,T) cannot be
    //    applied to (Holder<capture of ?>,Long))
    // wildSupertype(bounded, lng); // Błąd
    //    (wildSupertype(Holder<? super T>,T) cannot be
    //    applied to (Holder<capture of ? extends Long>,Long))
  }
} ///:~
```

W metodzie `rawArgs()` kompilator wie, że `Holder` to typ uogólniony, więc mimo że jest tam wyrażany jako goły typ, kompilator uznaje przekazywanie `Object` do wywołania `set()` za niebezpieczne. Skoro mamy do czynienia z gołym typem, możemy przekazywać do `set()` obiekty dowolnego typu, a będą one automatycznie rzutowane w górę na typ `Object`. Korzystanie z gołego typu oznacza więc rezygnację z kontroli w czasie kompilacji. To samo widać w wywołaniu `get()`: nie ma `T`, więc wynikiem może być jedynie `Object`.

Łatwo dać się zwieść pozorom i uznać, że `Holder` i `Holder<?>` to praktycznie to samo. Ale metoda `unboundedArg()` uwidacznia różnicę — powoduje te same problemy co `rawArg()`, ale tym razem są one wykrywane przez kompilator jako błędy, a nie ostrzeżenia, bo goły `Holder` może przechowywać kombinację dowolnych typów, a `Holder<?>` to heterogeniczna kolekcja obiektów *jakiegoś* (ale jednego) *typu*, w której nie można umieścić obiektu typu `Object`.

W metodach `exact1()` i `exact2()` wykorzystywane są konkretne parametry uogólnień — bez symboli wieloznacznych. Z powodu dodatkowego argumentu metoda `exact2()` różni się od `exact1()` zakresem ograniczeń.

W metodzie `wildSubtype()` ograniczenia odnośnie typu `Holder` są rozluźniane tak, że `Holder` może przechowywać cokolwiek, co jest pochodną (albo implementacją) `T`. Oznacza to znowu, że `T` mogłoby być typem `Fruit`, podczas gdy obiekt `holder` mógłby być typu `Holder<Apple>`. Wywołanie `set()` (i wszelkie inne metody przyjmujące argument typu określonego parametrem typowym uogólnienia) jest blokowane, co zapobiega umieszczeniu w kontenerze `Holder<Apple>` obiektów `Orange`. Wiadomo jednak, że wszelkie

obiekty znajdujące się w kontenerze są typu `Fruit` albo dowolnego typu pochodnego, można więc śmiało wywoływać metodę `get()` (i wszystkie inne zwracające wartości typu określonego parametrem typowym uogólnienia).

Wreszcie w metodzie `wildSupertype()` mamy do czynienia z zachowaniem odwrotnym do `wildSubtype()`: `holder` może tu być kontenerem przechowującym obiekty dowolnego typu bazowego względem `T`. Dlatego też metoda `set()` może przyjmować argumenty typu `T`, bo na mocy dziedziczenia wszystkie operacje na typie bazowym działają (polimorficznie) również dla typu pochodnego (tutaj jest nim właśnie `T`). Ale już wywołania `get()` są mało pomocne, bo typ obiektów przechowywanych w kontenerze `holder` może być dowolnym nadtypem `T`, więc jedynym bezpiecznym typem wartości zwracanej byłby `Object`.

Ten obszerny przykład ilustruje również ograniczenia operacji dotyczących parametru bez ramy konkretyzacji: z braku `T` nie można wywoływać ani `get()`, ani `set()`.

W metodzie `main()` można sprawdzić, które z tych metod akceptują poszczególne typy argumentów bez błędów i ostrzeżeń. Ze względu na zgodność migracji metoda `rawArgs()` będzie przyjmować wszelkie wariacje typu `Holder` bez słowa ostrzeżenia. Metoda `unboundedArg()` równie chętnie przyjmuje dowolne typy, ale — jako się rzekło — w ciele metody obsługuje je inaczej.

Jeśli do metody `exact1()`, przyjmującej „konkretny" (to jest bez symboli wieloznacznych) typ uogólnienia, przekazany zostanie obiekt gołego typu `Holder`, wywołanie takie zaowocuje ostrzeżeniem — argument powinien nieść informacje, których brak w gołym typie. Z kolei przekazanie referencji bez ramy konkretyzacji do metody `exact1()` oznacza brak informacji o typie potrzebnym do ustalenia typu wartości zwracanej.

Najwięcej ograniczeń narzuca metoda `exact2()`, bo wymaga przekazania obiektu typu `Holder<T>` i argumentu typu `T`, a w przypadku innych argumentów powoduje ostrzeżenia i błędy. Niekiedy jest to nawet pożądane, a w pozostałych przypadkach tak silne ograniczenie można złagodzić za pomocą symboli wieloznacznych — zależnie od tego, czy zamierzasz otrzymywać typowane wartości zwracane (jak w `wildSubtype()`) czy przekazywać typowane argumenty (jak w `wildSupertype()`).

Zaletą stosowania „konkretnych" parametrów typowych w miejsce symboli wieloznacznych jest większy zakres możliwości operacji na parametrach uogólnienia. Z kolei symbole wieloznaczne pozwalają na akceptowanie szerszego zestawu parametryzowanych typów w roli argumentów. Coś za coś, a decyzje o stosowaniu jednego bądź drugiego podejścia należy podejmować dla każdego przypadku indywidualnie.

Konwersja z przechwyceniem typu

Istnieje taka sytuacja, która wprost *wymaga* zastosowania symbolu wieloznacznego `<?>` w miejsce gołego typu. Przekazanie do metody operującej na `<?>` gołego typu umożliwia kompilatorowi dedukcję parametru typowego, dzięki czemu metoda może wywołać inną metodę z użyciem dokładnego typu. Technikę tę pokażę w kolejnym przykładzie; nosi ona miano konwersji z przechwyceniem, bo nieokreślony typ reprezentowany

symbolem wieloznacznym jest przechwytywany i konwertowany na typ dokładny. W poniższym programie komentarze dotyczące ostrzeżeń dotyczą kodu pozbawionego adnotacji `@SuppressWarnings`:

```
//: generics/CaptureConversion.java

public class CaptureConversion {
  static <T> void f1(Holder<T> holder) {
    T t = holder.get();
    System.out.println(t.getClass().getSimpleName());
  }
  static void f2(Holder<?> holder) {
    f1(holder); // Wywołanie z przechwyconym typem
  }
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    Holder raw = new Holder<Integer>(1);
    // f1(raw); // Powoduje ostrzeżenia
    f2(raw); // Bez ostrzeżeń
    Holder rawBasic = new Holder();
    rawBasic.set(new Object()); // Ostrzeżenie
    f2(rawBasic); // Bez ostrzeżeń
    // Rzutowanie w górę na Holder<?>, wciąż do wykrycia:
    Holder<?> wildcarded = new Holder<Double>(1.0);
    f2(wildcarded);
  }
} /* Output:
Integer
Object
Double
*///:~
```

Parametry typowe w `f1()` są jednoznaczne, bez symboli wieloznacznych. W metodzie `f2()` parametr `Holder` jest określony z symbolem wieloznacznym bez ramy konkretyzacji, więc w efekcie parametr typowy jest zupełnie nieznany. Ale w obrębie `f2()` następuje wywołanie `f1()`, a `f1()` wymaga przecież jednoznacznego parametru typowego. W procesie wywołania `f2()` odbywa się tu przechwycenie typu parametru, tak aby dało się go użyć w zagnieżdżonym tam wywołaniu `f1()`.

Jeśli zastanawiasz się nad przydatnością tej techniki przy zapisie, powinieneś pamiętać, że wraz z `Holder<?>` musiałbyś przekazać pewien typ. Konwersja z przechwyceniem sprawdza się tylko tam, gdzie w obrębie metody chcemy operować na jednoznacznym typie. Zauważ, że z metody `f2()` nie możemy zwrócić `T`, bo typ `T` jest w `f2()` nieznany. Konwersja z przechwyceniem jest więc ciekawa, ale mocno ograniczona.

Ćwiczenie 29. Utwórz metodę uogólnioną przyjmującą argument typu `Holder<List<?>>`. Określ metody, które możesz wywołać dla typów `Holder` i `List` oraz te metody, których wywołać nie możesz. Powtórz ćwiczenie dla argumentu typu `List<Holder<?>>` (5).

Problemy

Niniejszy podrozdział poświęcony jest rozmaitym problemom, z którymi przychodzi się borykać programistom korzystającym z uogólnień.

Typy podstawowe jako parametry typowe

W rozdziale padło już stwierdzenie, że jednym z ograniczeń uogólnień Javy jest niemożność nadawania parametrom typowym typów podstawowych. Nie można więc na przykład utworzyć kontenera `ArrayList<int>`.

Rozwiązaniem są oczywiście klasy opakowujące wartości typów podstawowych w połączeniu z mechanizmem automatycznego pakowania takich wartości w obiekty tych klas (w Javie SE5). Otóż jeśli utworzysz kontener typu `ArrayList<Integer>` i będziesz do niego dodawał wartości typu `int`, przekonasz się, że konwersja typu `int` na `Integer` odbywa się całkowicie automatycznie, a więc od strony praktycznej taki kontener nie różni się od `ArrayList<int>`:

```
//: generics/ListOfInt.java
// Mechanizm automatycznego pakowania w obiekty
// rekompensuje niemożność stosowania typów podstawowych
// do konkretyzacji uogólnień.
import java.util.*;

public class ListOfInt {
  public static void main(String[] args) {
    List<Integer> li = new ArrayList<Integer>();
    for(int i = 0; i < 5; i++)
      li.add(i);
    for(int i : li)
      System.out.print(i + " ");
  }
} /* Output:
0 1 2 3 4
*///:~
```

Ten sam wygodny mechanizm pozwala na generowanie wartości `int` w pętli foreach.

Zasadniczo rozwiązanie to działa świetnie: programista ma możliwość umieszczania wartości `int` w kontenerze i wydobywania ich z tego kontenera. Operacjom tym towarzyszy dodatkowy etap konwersji, ale jest ona dla programisty transparentna. Jeśli narzut konwersji powoduje nadmierny spadek wydajności, można skorzystać ze specjalizowanych odmian kontenerów przystosowanych do typów podstawowych; jedną z takich odmian jest `org.apache.commons.collections.primitives`.

Oto inne podejście, prezentowane na przykładzie tworzeniu zbioru (`Set`) wartości typu `Byte`:

```
//: generics/ByteSet.java
import java.util.*;
```

```
public class ByteSet {
  Byte[] possibles = { 1,2,3,4,5,6,7,8,9 };
  Set<Byte> mySet =
    new HashSet<Byte>(Arrays.asList(possibles));
  // Ale nie można tak:
  // Set<Byte> mySet2 = new HashSet<Byte>(
  //   Arrays.<Byte>asList(1,2,3,4,5,6,7,8,9));
} ///.~
```

Automatyczne pakowanie wartości podstawowych w obiekty odpowiedniego typu nie rozwiązuje wszystkich problemów. Poniższy przykład prezentuje uogólniony interfejs `Generator` z metodą `next()` zwracającą obiekt typu odpowiadającego parametrowi typowemu. Klasa `FArray` zawiera uogólnioną metodę wykorzystującą ten generator do wypełnienia tablicy obiektami (uogólnienie całej *klasy* nie zadziałałoby tu, bo rzeczona metoda ma być statyczna). Implementacje interfejsu `Generator` zaczerpniemy z rozdziału „Tablice"; w metodzie `main()` programu widnieje wywołanie `FArray.fill()` wypełniające tablice obiektami:

```
//: generics/PrimitiveGenericTest.java
import net.mindview.util.*;

// Wypełnienie tablicy za pomocą generatora:
class FArray {
  public static <T> T[] fill(T[] a, Generator<T> gen) {
    for(int i = 0; i < a.length; i++)
      a[i] = gen.next();
    return a;
  }
}

public class PrimitiveGenericTest {
  public static void main(String[] args) {
    String[] strings = FArray.fill(
      new String[7], new RandomGenerator.String(10));
    for(String s : strings)
      System.out.println(s);
    Integer[] integers = FArray.fill(
      new Integer[7], new RandomGenerator.Integer());
    for(int i: integers)
      System.out.println(i);
    // Automatyczne pakowanie w obiekty nic tu
    // nie pomoże. Kod nie skompiluje się:
    // int[] b =
    //     FArray.fill(new int[7], new RandIntGenerator());
  }
} /* Output:
YNzbrnyGcF
OWZnTcQrGs
eGZMmJMRoE
suEcUOneOE
dLsmwHLGEa
hKcxrEqUCB
bkInaMesbt
7052
6665
2654
3909
```

```
5202
2209
5458
*///.~
```

Ponieważ klasa `RandomGenerator.Integer` implementuje interfejs `Generator<Integer>`, można mieć nadzieję, że dzięki automatycznemu pakowaniu w obiekty dojdzie do automatycznej konwersji wartości zwracanej z metody `next()` z typu `Integer` na typ `int`. Niestety, mechanizm ten nie dotyczy tablic.

Ćwiczenie 30. Utwórz kontener `Holder` dla każdego typu opakowującego typ podstawowy i pokaż, że mechanizm pakowania w obiekty (i mechanizm odwrotny) działa skutecznie dla metod `set()` i `get()` wszystkich egzemplarzy `Holder` (2).

Implementowanie interfejsów parametryzowanych

Klasa nie może implementować dwóch wariantów tego samego interfejsu uogólnionego. Przyczyną jest oczywiście zacieranie, po którym obie konkretyzacje uogólnienia będą w istocie tym samym interfejsem. Oto przypadek takiej kolizji:

```
//: generics/MultipleInterfaceVariants.java
// {CompileTimeError} (Nie skompiluje się)

interface Payable<T> {}

class Employee implements Payable<Employee> {}
class Hourly extends Employee
  implements Payable<Hourly> {} ///:~
```

Klasa `Hourly` nie skompiluje się, bo zacieranie zredukuje interfejsy `Payable<Employee>` i `Payable<Hourly>` do tej samej klasy `Payable`, co w powyższym kodzie oznaczałoby dwukrotną implementację tego samego interfejsu w jednej klasie. Co ciekawe, gdybyś usunął parametry uogólnienia z obu miejsc użycia `Payable` — jak to czyni kompilator w ramach zacierania — kod dałby się skompilować.

Ten problem daje się we znaki, kiedy operuje się na niektórych podstawowych interfejsach Javy, jak `Comparable<T>`, co pokażę w dalszej części tego podrozdziału.

Ćwiczenie 31. Usuń uogólnienia z programu *MultipleInterfaceVariants.java* i zmodyfikuj kod tak, aby program dał się skompilować (1).

Ostrzeżenia przy rzutowaniu

Użycie rzutowania albo operatora `instanceof` z parametrem typowym uogólnienia nie przynosi żadnego efektu. Poniższy kontener wewnętrznie przechowuje elementy jako egzemplarze klasy `Object`, dokonując rzutowania na typ `T` w ramach wydobywania elementów z kontenera:

```
//: generics/GenericCast.java

class FixedSizeStack<T> {
  private int index = 0;
```

```
  private Object[] storage;
  public FixedSizeStack(int size) {
    storage = new Object[size];
  }
  public void push(T item) { storage[index++] = item; }
  @SuppressWarnings("unchecked")
  public T pop() { return (T)storage[--index]; }
}

public class GenericCast {
  public static final int SIZE = 10;
  public static void main(String[] args) {
    FixedSizeStack<String> strings =
      new FixedSizeStack<String>(SIZE);
    for(String s : "A B C D E F G H I J".split(" "))
      strings.push(s);
    for(int i = 0; i < SIZE; i++) {
      String s = strings.pop();
      System.out.print(s + " ");
    }
  }
} /* Output:
J I H G F E D C B A
*///:~
```

Bez adnotacji `@SuppressWarnings` kompilator wygenerowałby ostrzeżenie o „niesprawdzanym rzutowaniu" przy metodzie `pop()`. Z racji zacierania kompilator nie wie, czy rzutowanie jest bezpieczne, a metoda `pop()` tak naprawdę nie realizuje żadnego rzutowania. Typ `T` jest zacierany do pierwszej ramy, którą jest domyślnie typ `Object`, więc `pop()` w zasadzie rzutuje typ `Object` na typ `Object`.

Nie zawsze typ uogólniony eliminuje potrzebę rzutowania, co powoduje niewłaściwy komunikat ostrzeżenia ze strony kompilatora. Oto przykład:

```
//: generics/NeedCasting.java
import java.io.*;
import java.util.*;

public class NeedCasting {
  @SuppressWarnings("unchecked")
  public void f(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream(args[0]));
    List<Widget> shapes = (List<Widget>)in.readObject();
  }
} ///:~
```

W następnym rozdziale przekonasz się, że metoda `readObject()` nie może wiedzieć, co odczytuje, więc zwraca obiekt wymagający rzutowania. Ale po oznaczeniu jako komentarz adnotacji `@SuppressWarnings` i ponownej kompilacji programu otrzymamy ostrzeżenie:

```
Note: NeedCasting.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details
```

A jeśli skorzystamy z porady i wykonamy kompilację z opcją `-Xlint:unchecked`, otrzymamy coś takiego:

```
NeedCasting.java:10: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: java.util.List<Widget>
    List<Widget> shapes = (List<Widget>)in.readObject();
```

Trzeba rzutować, a kompilator nie pozwala. Aby rozwiązać powstały problem, należy skorzystać z formy rzutowania wprowadzonej w Javie SE5, czyli rzutowania przez klasę uogólnioną:

```
//: generics/ClassCasting.java
import java.io.*;
import java.util.*;

public class ClassCasting {
  @SuppressWarnings("unchecked")
  public void f(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream(args[0]));
      // Nie skompiluje się:
//    List<Widget> lw1 =
//    List<Widget>.class.cast(in.readObject());
    List<Widget> lw2 = List.class.cast(in.readObject());
  }
} ///:~
```

Nie można jednak rzutować na właściwy typ `List<Widget>`. Nie można zapisać czegoś takiego:

```
List<Widget>.class.cast(in.readObject());
```

a nawet tak (z dodatkowym rzutowaniem):

```
(List<Widget>)List.class.cast(in.readObject());
```

— oba zapisy spowodują ostrzeżenie kompilatora.

Ćwiczenie 32. Sprawdź, czy klasa `FixedSizeStack` z programu *GenericCast.java* powoduje wyjątki przy próbie wyjścia poza ramy. Czy to oznacza, że nie trzeba stosować kodu kontrolującego wykraczanie poza ramy (1)?

Ćwiczenie 33. Popraw program *GenericCast.java*, korzystając z kontenera `ArrayList` (3).

Przeciążanie

Poniższy kod, choć wcale sensowny, nie skompiluje się:

```
//: generics/UseList.java
// {CompileTimeError} (Nie skompiluje się)
import java.util.*;

public class UseList<W,T> {
  void f(List<T> v) {}
  void f(List<W> v) {}
} ///:~
```

Przeciążanie metody wprowadza do programu sygnaturę identyczną z już istniejącą — to efekt zacierania argumentów konkretyzacji uogólnienia.

Zamiast przeciążania trzeba zastosować różnicowanie nazw metod tak, aby zacieranie typów w listach argumentów wywołań nie powodowało ujednolicania sygnatur:

```
//: generics/UseList2.java
import java.util.*;

public class UseList2<W,T> {
  void f1(List<T> v) {}
  void f2(List<W> v) {}
} ///:~
```

Na szczęście problemy tego rodzaju są wykrywane przez kompilator.

Zawłaszczenie interfejsu w klasie bazowej

Załóżmy, że dysponujemy klasą Pet implementującą interfejs Comparable, uzdatniający obiekt klasy Pet do porównywania z innymi obiektami tej klasy:

```
//: generics/ComparablePet.java

public class ComparablePet
implements Comparable<ComparablePet> {
  public int compareTo(ComparablePet arg) { return 0; }
} ///:~
```

Pożądana byłaby tu próba ograniczenia typów, z którymi można by porównywać podklasy ComparablePet. Na przykład podtyp Cat powinien być porównywalny jedynie z innymi podtypami Cat:

```
//: generics/HijackedInterface.java
// {CompileTimeError} (Nie skompiluje się)

class Cat extends ComparablePet implements Comparable<Cat>{
  // Błąd: "Comparable cannot be inherited with
  // different arguments: <Cat> and <Pet>"
  public int compareTo(Cat arg) { return 0; }
} ///:~
```

Niestety, to nie zadziała. Kiedy argument ComparablePet zostanie ustalony dla Comparable, żadna inna klasa implementująca interfejs nie będzie mogła być porównywana z czymkolwiek poza ComparablePet:

```
//: generics/RestrictedComparablePets.java

class Hamster extends ComparablePet
implements Comparable<ComparablePet> {
  public int compareTo(ComparablePet arg) { return 0; }
}

// albo po prostu:

class Gecko extends ComparablePet {
  public int compareTo(ComparablePet arg) { return 0; }
} ///:~
```

Klasa Hamster pokazuje, że można ponownie zaimplementować ten sam interfejs, który implementuje ComparablePet, pod warunkiem, że jest on dokładnie taki sam, z nazwami typów parametrów włącznie. Ale wtedy możemy równie dobrze po prostu przeciążyć metody interfejsu z klasy bazowej, jak to zostało zrobione w klasie Gecko.

Typy samoskierowane

W omówieniach typów uogólnionych w Javie pojawia się często coś przypominającego szaradę. Wygląda to tak:

```
class SelfBounded<T extends SelfBounded<T>> { // ...
```

To jakby ustawić naprzeciw siebie dwa lustra, z efektem odbicia powtarzanego w nieskończoność. Klasa SelfBounded przyjmuje argument konkretyzacji uogólnienia T, który jest ograniczony ramą, a jest nią typ SelfBounded z T w roli argumentu.

Taki zapis jest na pierwszy rzut oka trudny do ogarnięcia; tu najlepiej widać, że słowo kluczowe extends stosowane w ramach uogólnień ma znaczenie zupełnie inne niż przy tworzeniu klas pochodnych.

Osobliwa rekurencja uogólnienia

Aby zrozumieć, co oznacza taka rekurencja, należałoby zacząć od najprostszej wersji tego idiomu, pozbawionej jeszcze skierowania zwrotnego.

Otóż nie można dziedziczyć wprost po parametrze uogólnienia, ale *można* dziedziczyć po klasie, która korzysta z parametru uogólnionego w jej własnej definicji. Można więc zapisać:

```
//: generics/CuriouslyRecurringGeneric.java

class GenericType<T> {}

public class CuriouslyRecurringGeneric
  extends GenericType<CuriouslyRecurringGeneric> {} ///:~
```

Taki kod można określić mianem *osobliwej rekurencji uogólnienia* (CRG, od *curiously recurring generics*), za ukutym przez Jima Copliena podobnym terminem wzorca osobliwej rekurencji szablonu (CRTP, od *curiously recurring template pattern*) w języku C++. Pojęcie „osobliwej rekurencji" odnosi się do faktu (dość osobliwego) występowania danej klasy w jej własnej klasie bazowej.

Aby zrozumieć znaczenie takiej rekurencji, najlepiej głośno powtórzyć sobie sens zapisu: „tworzę nową klasę wyprowadzoną z typu uogólnionego, który przyjmuje tę nową klasę w roli parametru uogólnienia". Co może zrobić uogólniona klasa bazowa z nazwą typu pochodnego? Cóż, uogólnienia w Javie dotyczą argumentów wywołań i wartości zwracanych metod, więc może to posłużyć do utworzenia klasy bazowej, która wykorzystuje typ klasy pochodnej w argumentach wywołań metod i ich wartości zwracanych.

Co więcej, taka uogólniona klasa bazowa może wykorzystać typ pochodny do definiowania typów pól obiektu, mimo że potem typ ten zostanie zatarty do postaci typu Object. Oto przykład takiej klasy:

```
//: generics/BasicHolder.java

public class BasicHolder<T> {
  T element;
  void set(T arg) { element = arg; }
  T get() { return element; }
  void f() {
    System.out.println(element.getClass().getSimpleName());
  }
} ///:~
```

To najzwyklejszy typ uogólniony z metodami przyjmującymi i zwracającymi obiekty typu konkretyzującego uogólnienie oraz z metodą operującą na polu tego typu (choć operacje te ograniczają się do operacji dozwolonych dla obiektów klasy Object).

Klasę BasicHolder możemy wykorzystać w osobliwej rekurencji uogólnienia:

```
//: generics/CRGWithBasicHolder.java

class Subtype extends BasicHolder<Subtype> {}

public class CRGWithBasicHolder {
  public static void main(String[] args) {
    Subtype st1 = new Subtype(), st2 = new Subtype();
    st1.set(st2);
    Subtype st3 = st1.get();
    st1.f();
  }
} /* Output:
Subtype
*///:~
```

Zwróć uwagę na coś istotnego: nowa klasa Subtype przyjmuje i zwraca wartości typu Subtype, a nie po prostu klasy bazowej BasicHolder. To właśnie sedno CRG: *klasa bazowa podstawia w miejsce parametrów klasy pochodne*. Oznacza to, że uogólniona klasa bazowa staje się czymś w rodzaju szablonu wspólnego zestawu funkcji dla wszystkich klas pochodnych, ale funkcje te operują w zakresie argumentów wywołań i wartości zwracanych na typach pochodnych. W klasie pochodnej wykorzystywany będzie typ tejże klasy, a nie typ klasy bazowej. Dlatego w klasie Subtype oba argumenty metody set() i typ wartości zwracanej metody get() to obiekty typu Subtype.

Samoskierowanie

Klasa BasicHolder może wykorzystywać do konkretyzacji parametru typowego uogólnienia dowolne typy, jak tu:

```
//: generics/Unconstrained.java

class Other {}
class BasicOther extends BasicHolder<Other> {}
```

```
public class Unconstrained {
  public static void main(String[] args) {
    BasicOther b = new BasicOther(), b2 = new BasicOther();
    b.set(new Other());
    Other other = b.get();
    b.f();
  }
} /* Output:
Other
*///:~
```

Samoskierowanie polega na dodatkowym zmuszeniu typu ogólnego, by stał się własnym argumentem ramy. Zobaczmy, jak to się robi i jak można wykorzystać otrzymaną tak klasę:

```
//: generics/SelfBounding.java

class SelfBounded<T extends SelfBounded<T>> {
  T element;
  SelfBounded<T> set(T arg) {
    element = arg;
    return this;
  }
  T get() { return element; }
}

class A extends SelfBounded<A> {}
class B extends SelfBounded<A> {} // Też dobrze

class C extends SelfBounded<C> {
  C setAndGet(C arg) { set(arg); return get(); }
}

class D {}
// Nie można tak:
// class E extends SelfBounded<D> {}
// Błąd kompilacji: parametr typowy D spoza ramy

// Niestety, można też tak, więc nie można narzucić idiomu:
class F extends SelfBounded {}

public class SelfBounding {
  public static void main(String[] args) {
    A a = new A();
    a.set(new A());
    a = a.set(new A()).get();
    a = a.get();
    C c = new C();
    c = c.setAndGet(new C());
  }
} ///:~
```

Samoskierowanie oznacza wymuszenie użycia klasy w relacji dziedziczenia takiej jak poniżej:

```
class A extends Selfbounded<A> {}
```

Wymusza to przekazanie definiowanej klasy w roli argumentu konkretyzacji uogólnienia klasy bazowej.

Co pozytywnego wynika z takiego idiomu? Otóż parametr typowy klasy bazowej musi być taki sam jak definiowana klasa pochodna. W definicji klasy B widać, że można też wyprowadzać pochodną z klasy SelfBounded parametryzowanej klasą pochodną inną niż B (tu A), choć tego rodzaju zastosowania stanowią margines. Próba definiowania typu E pokazuje zaś, że nie można wykorzystać parametru typowego spoza klasy (hierarchii klas) SelfBounded.

Niestety, klasa F daje się skompilować bez ostrzeżeń, więc idiomu samoskierowania nie da się narzucić programistom-użytkownikom. Tam, gdzie jest to ważne, można skorzystać z zewnętrznego narzędzia, które zapewni, że w miejsce typów parametryzowanych nie będą wykorzystywane typy surowe.

Zauważ, że możesz usunąć ograniczenie, nie blokując możliwości kompilacji wszystkich przykładowych klas, ale wtedy skompiluje się również klasa E:

```
//: generics/NotSelfBounded.java

public class NotSelfBounded<T> {
  T element;
  NotSelfBounded<T> set(T arg) {
    element = arg;
    return this;
  }
  T get() { return element; }
}

class A2 extends NotSelfBounded<A2> {}
class B2 extends NotSelfBounded<A2> {}

class C2 extends NotSelfBounded<C2> {
  C2 setAndGet(C2 arg) { set(arg); return get(); }
}

class D2 {}
// Teraz w porządku:
class E2 extends NotSelfBounded<D2> {} ///:~
```

Najwyraźniej samoskierowanie przydaje się jedynie przy narzucaniu relacji dziedziczenia. Przy zastosowaniu samoskierowania wiadomo, że parametr typowy używany w klasie będzie tego samego typu co klasa używająca parametru.

Samoskierowanie można zastosować również w metodach uogólnionych:

```
//: generics/SelfBoundingMethods.java

public class SelfBoundingMethods {
  static <T extends SelfBounded<T>> T f(T arg) {
    return arg.set(arg).get();
  }
  public static void main(String[] args) {
    A a = f(new A());
  }
} ///:~
```

W ten sposób metoda jest zabezpieczana przed stosowaniem wobec czegokolwiek, co nie pochodzi z hierarchii typu samoskierowanego SelfBounded.

Kowariancja argumentów

Wartością typów samoskierowanych jest dawana przez nie możliwość tworzenia *kowariantnych typów argumentów* — typów argumentów metod zmiennych w zależności od położenia metody w hierarchii klas.

Choć typy samoskierowane tworzą również wartości zwracane typów zgodnych z typami podklas, nie ma to aż takiego znaczenia, bo w Javie SE5 wprowadzona została *kowariancja typów zwracanych*:

```
//: generics/CovariantReturnTypes.java

class Base {}
class Derived extends Base {}

interface OrdinaryGetter {
  Base get();
}

interface DerivedGetter extends OrdinaryGetter {
  // Typ zwracany metody przesłanianej może się zmieniać:
  Derived get();
}

public class CovariantReturnTypes {
  void test(DerivedGetter d) {
    Derived d2 = d.get();
  }
} ///:~
```

Metoda `get()` w klasie `DerivedGetter` przesłania wersję `get()` z klasy `OrdinaryGetter`, a przy tym zwraca typ wyprowadzony z typu zwracanego przez wywołanie `OrdinaryGetter.get()`. To rzecz jak najbardziej sensowna — wszak metoda typu pochodnego powinna mieć możliwość zwracania typu specjalizowanego względem typu zwracanego przez przesłanianą metodę typu bazowego — ale w poprzednich wydaniach Javy było to niedozwolone.

Uogólnienie z samoskierowaniem również wykorzystuje dokładny typ pochodny w roli typu wartości zwracanej, jak w poniższej wersji `get()`:

```
//: generics/GenericsAndReturnTypes.java

interface GenericGetter<T extends GenericGetter<T>> {
  T get();
}

interface Getter extends GenericGetter<Getter> {}

public class GenericsAndReturnTypes {
  void test(Getter g) {
    Getter result = g.get();
```

```
    GenericGetter gg = g.get(); // Również typ bazowy
  }
} ///:~
```

Zauważ, że taki kod nie skompilowałby się, gdyby nie kowariancja typów wartości zwracanych wprowadzona w Javie SE5.

Jednakże w kodzie nieuogólnionym argumenty wywołań metod *nie mogą zmieniać* się wraz z położeniem metod w hierarchii klas:

```
//: generics/OrdinaryArguments.java

class OrdinarySetter {
  void set(Base base) {
    System.out.println("OrdinarySetter.set(Base)");
  }
}

class DerivedSetter extends OrdinarySetter {
  void set(Derived derived) {
    System.out.println("DerivedSetter.set(Derived)");
  }
}

public class OrdinaryArguments {
  public static void main(String[] args) {
    Base base = new Base();
    Derived derived = new Derived();
    DerivedSetter ds = new DerivedSetter();
    ds.set(derived);
    ds.set(base); // Kompiluje się: mamy przeciążenie, nie przesłonięcie!
  }
} /* Output:
DerivedSetter.set(Derived)
OrdinarySetter.set(Base)
*///:~
```

Na rzecz obiektu klasy pochodnej można równie dobrze wywołać `set(derived)` i `set(base)`, co oznacza, że nie mamy tu do czynienia z przesłonięciem metody, a jedynie z jej przeciążeniem dla dwóch różnych typów. Na wyjściu programu widać, że klasa `DerivedSetter` posiada dwie metody set, a więc dysponuje wciąż wersją odziedziczoną po klasie bazowej, co utwierdza w przekonaniu, że nie doszło do jej przesłonięcia, a jedynie do przeciążenia.

Ale w typach samoskierowanych w klasie pochodnej widnieje tylko jedna wersja metody, i to taka, która przyjmuje argument typu pochodnego, a nie typu bazowego:

```
//: generics/SelfBoundingAndCovariantArguments.java

interface SelfBoundSetter<T extends SelfBoundSetter<T>> {
  void set(T arg);
}

interface Setter extends SelfBoundSetter<Setter> {}

public class SelfBoundingAndCovariantArguments {
```

```
  void testA(Setter s1, Setter s2, SelfBoundSetter sbs) {
    s1.set(s2);
    // s1.set(sbs); // Błąd:
    // set(Setter) in SelfBoundSetter<Setter>
    // cannot be applied to (SelfBoundSetter)
  }
} ///:~
```

Kompilator nie rozpoznaje próby przekazania typu bazowego w roli argumentu wywołania metody `set()`, bo nie zna metody o takiej sygnaturze. Doszło niniejszym do przesłonięcia argumentu.

Bez samoskierowania do gry wkroczyłby zwyczajny mechanizm dziedziczenia i otrzymalibyśmy przeciążenie, tak jak w przypadku bez uogólnienia:

```
//: generics/PlainGenericInheritance.java

class GenericSetter<T> { // Bez samoskierowania
  void set(T arg){
    System.out.println("GenericSetter.set(Base)");
  }
}

class DerivedGS extends GenericSetter<Base> {
  void set(Derived derived){
    System.out.println("DerivedGS.set(Derived)");
  }
}

public class PlainGenericInheritance {
  public static void main(String[] args) {
    Base base = new Base();
    Derived derived = new Derived();
    DerivedGS dgs = new DerivedGS();
    dgs.set(derived);
    dgs.set(base); // Kompiluje się: mamy przeciążenie, nie przesłonięcie!
  }
} /* Output:
DerivedGS.set(Derived)
GenericSetter.set(Base)
*///:~
```

Kod ten naśladuje program *OrdinaryArguments.java*; w tamtym przykładzie klasa `DerivedSetter` dziedziczyła po `OrdinarySetter`, która zawierała metodę `set(Base)`. Tutaj `DerivedGS` dziedziczy po `GenericSetter<Base>`, która również zawiera metodę `set(Base)` utworzoną w ramach uogólnienia. I tak jak w programie *OrdinaryArguments.java*, na wyjściu programu widać, że `DerivedGS` zawiera dwie przeciążone wersje metody `set()`. Bez samoskierowania przeciążamy metody dla typów argumentów. Z samoskierowaniem otrzymujemy jedną wersję metody, z typem argumentu odpowiadającym typowi podklasy zawierającej metodę.

Ćwiczenie 34. Utwórz samoskierowany typ uogólniony zawierający metodę abstrakcyjną przyjmującą argument typu uogólnionego i zwracającą wartość typu uogólnionego. W nieabstrakcyjnej metodzie tej klasy wywołaj metodę abstrakcyjną i zwróć jej wynik do wywołującego. Wyprowadź klasę pochodną typu samoskierowanego i przetestuj jej zachowanie (4).

Dynamiczna kontrola typów

Ponieważ programiści mają możliwość przekazywania kontenerów uogólnionych do kodu przygotowanego pod kątem wcześniejszych wersji Javy, istnieje ryzyko, że ów kod uszkodzi taki kontener. W takich sytuacjach można skorzystać z narzędzi kontroli typów udostępnianych w bibliotece `java.util.Collections` w wydaniu SE5 — chodzi o statyczne metody `checkedCollection()`, `checkedList()`, `checkedMap()`, `checkedSet()`, `checkedSortedMap()` i `checkedSortedSet()`. Każda z nich przyjmuje za pośrednictwem pierwszego argumentu wywołania kontener, który ma podlegać dynamicznej kontroli typów, i pożądany typ (przekazywany drugim argumentem wywołania).

Kontener podlegający kontroli typów zgłosi wyjątek `ClassCastException` w momencie próby *umieszczenia* w nim obiektu nieodpowiedniego typu — w zwykłych (nieuogólnionych) kontenerach o problemie dowiedzielibyśmy się dopiero przy *wyjmowaniu* obiektu z kontenera. W takim przypadku wiedza o problemie nie niesie ze sobą wiedzy o jego źródle (winowajcy); w przypadku kontenerów kontrolowanych dynamicznie od razu wiadomo, kto próbuje wstawić niewłaściwy obiekt.

Przyjrzyjmy się zachowaniu kontenerów z dynamiczną kontrolą typów na przykładzie „umieszczania kota na liście psów". W omawianym kodzie metoda `oldStyleMethod()` reprezentuje zastany kod, który przyjmuje kontener surowego typu `List`; adnotacja `@SuppressWarnings` służy do tłumienia niepotrzebnych komunikatów ostrzeżeń:

```
//: generics/CheckedList.java
// Zastosowanie metody Collection.checkedList().
import typeinfo.pets.*;
import java.util.*;

public class CheckedList {
  @SuppressWarnings("unchecked")
  static void oldStyleMethod(List probablyDogs) {
    probablyDogs.add(new Cat());
  }
  public static void main(String[] args) {
    List<Dog> dogs1 = new ArrayList<Dog>();
    oldStyleMethod(dogs1); // Ciche przyjęcie obiektu Cat
    List<Dog> dogs2 = Collections.checkedList(
      new ArrayList<Dog>(), Dog.class);
    try {
      oldStyleMethod(dogs2); // Zgłoszenie wyjątku
    } catch(Exception e) {
      System.out.println(e);
    }
    // Działa dla typów pochodnych:
    List<Pet> pets = Collections.checkedList(
      new ArrayList<Pet>(), Pet.class);
    pets.add(new Dog());
    pets.add(new Cat());
  }
} /* Output:
java.lang.ClassCastException: Attempt to insert class typeinfo.pets.Cat element into collection with element
type class typeinfo.pets.Dog
*///:~
```

Po uruchomieniu programu zauważysz, że operacja wstawienia obiektu Cat do kontenera dogs1 przebiega bez echa, ale już wstawienie takiego obiektu do kontenera dogs2 powoduje zgłoszenie wyjątku w reakcji na niepoprawny typ elementu kontenera. Przekonasz się też, że możesz swobodnie umieszczać w kontenerze podlegającym kontroli dynamicznej typów typy pochodne względem typu określonego w deklaracji kontenera.

Ćwiczenie 35. Zmodyfikuj program *CheckedList.java* tak, aby wykorzystywał hierarchię klas Coffee zdefiniowaną wcześniej w rozdziale (1).

Wyjątki

Z powodu zacierania typów zastosowanie wyjątków w uogólnieniach jest wyjątkowo ograniczone. Klauzula catch nie może przechwytywać wyjątków typu uogólnionego, bo typ wyjątku (dokładny typ) musi być znany już na etapie kompilacji, a potem również w czasie wykonania. Ponadto klasa uogólniona nie może ani wprost, ani pośrednio dziedziczyć po klasie Throwable, co uniemożliwia z kolei próby definiowania wyjątków uogólnionych (których zresztą i tak nie dałoby się przechwytywać).

Jednakże w deklaracjach throws można odwoływać się do parametrów typowych występujących w deklaracji metody. Pozwala to na pisanie kodu uogólnionego różnicującego typ kontrolowanego wyjątku:

```
//: generics/ThrowGenericException.java
import java.util.*;

interface Processor<T,E extends Exception> {
  void process(List<T> resultCollector) throws E;
}

class ProcessRunner<T,E extends Exception>
extends ArrayList<Processor<T,E>> {
  List<T> processAll() throws E {
    List<T> resultCollector = new ArrayList<T>();
    for(Processor<T,E> processor : this)
      processor.process(resultCollector);
    return resultCollector;
  }
}

class Failure1 extends Exception {}

class Processor1 implements Processor<String,Failure1> {
  static int count = 3;
  public void
  process(List<String> resultCollector) throws Failure1 {
    if(count-- > 1)
      resultCollector.add("Hej!");
    else
      resultCollector.add("Ho!");
    if(count < 0)
      throw new Failure1();
  }
```

```
}

class Failure2 extends Exception {}

class Processor2 implements Processor<Integer,Failure2> {
  static int count = 2;
  public void
  process(List<Integer> resultCollector) throws Failure2 {
    if(count-- == 0)
      resultCollector.add(47);
    else {
      resultCollector.add(11);
    }
    if(count < 0)
       throw new Failure2();
  }
}

public class ThrowGenericException {
  public static void main(String[] args) {
    ProcessRunner<String,Failure1> runner =
      new ProcessRunner<String,Failure1>();
    for(int i = 0; i < 3; i++)
      runner.add(new Processor1());
    try {
      System.out.println(runner.processAll());
    } catch(Failure1 e) {
      System.out.println(e);
    }

    ProcessRunner<Integer,Failure2> runner2 =
      new ProcessRunner<Integer,Failure2>();
    for(int i = 0; i < 3; i++)
      runner2.add(new Processor2());
    try {
      System.out.println(runner2.processAll());
    } catch(Failure2 e) {
      System.out.println(e);
    }
  }
} ///:~
```

Klasa `Processor` wykonuje metodę `process()` i może zgłosić wyjątek typu `E`. Wynik metody `process()` jest składowany w `List<T> resultCollector` (to tak zwany *parametr zbierający*). Klasa `ProcessRunner` posiada z kolei metodę `processAll()` uruchamiającą poszczególne egzemplarze klasy `Processor` i zwracającą wypełniony kontener `resultCollector`.

Gdyby nie możliwość parametryzowania zgłaszanych wyjątków, nie można byłoby pisać takiego kodu w sposób uogólniony właśnie z powodu ścisłej kontroli typów wyjątków.

Ćwiczenie 36. Dodaj do klasy `Processor` drugi wyjątek sparametryzowany i pokaż, że jego typ może się zmieniać niezależnie od typu pierwszego wyjątku (2).

Domieszki

Pojęcie *domieszki* (*klasy mieszanej*) zyskało z czasem liczne znaczenia, ale zasadniczo chodzi o możliwość kombinowania cech wielu klas w celu stworzenia klasy reprezentującej wszystkie typy klasy mieszanej. To często czynność wykonywana na ostatnią chwilę, stąd wygoda i łatwość składania klas.

Zaletą klas mieszanych jest ujednolicenie cech i zachowania wielu klas. Dodatkowa korzyść polega na tym, że zmiana czegokolwiek w klasie domieszki jest od razu odzwierciedlana we wszystkich klasach zawierających domieszkę. Domieszki stanowią więc *odmianę programowania aspektowego*.

Domieszki w C++

W języku C++ domieszki stanowią jeden z silniejszych argumentów za wielokrotnym dziedziczeniem. Jednak bardziej eleganckim rozwiązaniem jest zastosowanie domieszek za pośrednictwem parametryzacji typów, gdzie domieszkę stanowi klasa wyprowadzona z parametru typowego. W języku C++ domieszki tworzy się prosto i łatwo, bo C++ zapamiętuje typy argumentów konkretyzujących uogólnienia.

Oto zaczerpnięty z języka C++ przykład z dwoma typami domieszek: jeden pozwala na wmieszanie do klasy cechy posiadania znacznika czasowego, drugi wyposaża każdy egzemplarz typu w jego własny numer seryjny:

```
//: generics/Mixins.cpp
#include <string>
#include <ctime>
#include <iostream>
using namespace std;

template<class T> class TimeStamped : public T {
  long timeStamp;
public:
  TimeStamped() { timeStamp = time(0); }
  long getStamp() { return timeStamp; }
};

template<class T> class SerialNumbered : public T {
  long serialNumber;
  static long counter;
public:
  SerialNumbered() { serialNumber = counter++; }
  long getSerialNumber() { return serialNumber; }
};

// Definicja i inicjalizacja zmiennej statycznej:
template<class T> long SerialNumbered<T>::counter = 1;

class Basic {
  string value;
public:
  void set(string val) { value = val; }
  string get() { return value; }
```

```
};

int main() {
  TimeStamped<SerialNumbered<Basic> > mixin1, mixin2;
  mixin1.set("ciąg testowy 1");
  mixin2.set("ciąg testowy 2");
  cout << mixin1.get() << " " << mixin1.getStamp() <<
    " " << mixin1.getSerialNumber() << endl;
  cout << mixin2.get() << " " << mixin2.getStamp() <<
    " " << mixin2.getSerialNumber() << endl;
} /* Output: (Sample)
ciąg testowy 1 1129840250 1
ciąg testowy 2 1129840250 2
*///:~
```

Powstające w metodzie `main()` egzemplarze `mixin1` i `mixin2` posiadają komplet metod właściwych dla obu mieszanych typów. Klasę mieszaną można uznać za funkcję odwzorowującą istniejące klasy na nowe podklasy. Zauważ, że utworzenie klasy mieszanej jest w takim układzie banalne: wystarczy wyrazić taką chęć wprost w kodzie:

```
TimeStamped<SerialNumbered<Basic> > mixin1, mixin2;
```

Niestety, uogólnienia w języku Java nie pozwalają na taką wygodę. Zacieranie typów powoduje „zapominanie" typu klasy bazowej, więc klasa uogólniona nie może dziedziczyć wprost po parametrze typowym uogólnienia.

Domieszki z użyciem interfejsów

Powszechnie proponowanym rozwiązaniem omawianego problemu tworzenia klas mieszanych w języku Java jest zastosowanie interfejsów, jak tutaj:

```
//: generics/Mixins.java
import java.util.*;

interface TimeStamped { long getStamp(); }

class TimeStampedImp implements TimeStamped {
  private final long timeStamp;
  public TimeStampedImp() {
    timeStamp = new Date().getTime();
  }
  public long getStamp() { return timeStamp; }
}

interface SerialNumbered { long getSerialNumber(); }

class SerialNumberedImp implements SerialNumbered {
  private static long counter = 1;
  private final long serialNumber = counter++;
  public long getSerialNumber() { return serialNumber; }
}

interface Basic {
  public void set(String val);
  public String get();
}
```

```
class BasicImp implements Basic {
  private String value;
  public void set(String val) { value = val; }
  public String get() { return value; }
}

class Mixin extends BasicImp
implements TimeStamped, SerialNumbered {
  private TimeStamped timeStamp = new TimeStampedImp();
  private SerialNumbered serialNumber =
    new SerialNumberedImp();
  public long getStamp() { return timeStamp.getStamp(); }
  public long getSerialNumber() {
    return serialNumber.getSerialNumber();
  }
}

public class Mixins {
  public static void main(String[] args) {
    Mixin mixin1 = new Mixin(), mixin2 = new Mixin();
    mixin1.set("ciąg testowy 1");
    mixin2.set("ciąg testowy 2");
    System.out.println(mixin1.get() + " " +
      mixin1.getStamp() +  " " + mixin1.getSerialNumber());
    System.out.println(mixin2.get() + " " +
      mixin2.getStamp() +  " " + mixin2.getSerialNumber());
  }
} /* Output: (Sample)
ciąg testowy 1 1132437151359 1
ciąg testowy 2 1132437151359 2
*///:~
```

Klasa `Mixins` opiera się w zasadzie na *delegacji*, więc każdy domieszkowany typ wymaga reprezentowania stosownym polem w klasie `Mixins`, a samą klasę trzeba wyposażyć we wszystkie metody delegujące wywołania do odpowiednich obiektów. W tym przykładzie prezentowane klasy są trywialne, ale w przypadku bardziej złożonych klas ilość kodu obsługującego domieszki z delegacjami może gwałtownie wzrosnąć[4].

Ćwiczenie 37. Dodaj do programu *Mixins.java* nową klasę `Colored`, wmieszaj ją do klasy `Mixins` i pokaż, że domieszka działa (2).

Zastosowanie wzorca projektowego Decorator

Jeśli przyjrzeć się zastosowaniom domieszek, przywodzą one na myśl dziedzinę zastosowań wzorca projektowego *Decorator* (dekorator)[5]. Dekorację stosuje się tam, gdzie wyprowadzanie podklas dla wszystkich możliwych kombinacji owocowałoby najzwyczajniej przerostem klas.

[4] Niektóre środowiska programistyczne, jak Eclipse czy IntelliJ Idea, automatycznie generują stosowny kod delegacji.

[5] Wzorce projektowe omawiam osobno w książce *Thinking in Patterns (with Java)* udostępnianej pod adresem www.MindView.net. Polecam też lekturę klasyki: *Design Patterns* Ericha Gammy i reszty (Addison-Wesley, 1995).

Omawiany wzorzec projektowy zakłada wykorzystanie warstw obiektów w celu dynamicznego i transparentnego uzupełniania poszczególnych obiektów o kolejne zadania. Dekoracja wymaga, aby wszystkie obiekty uzupełniające pierwotny obiekt posiadały ten sam podstawowy interfejs. Coś daje się dekorować, a my nakładamy na to funkcje, dekorując to coś klasami implementującymi funkcje. W ten sposób dekoracja jest transparentna — mamy zestaw wspólnych komunikatów wysyłanych do obiektu niezależnie od tego, czy (i czym) został „udekorowany". Klasa dekoracji może dodawać swoje metody, ale całość jest ograniczona.

Klasy dekoracji są implementowane za pomocą kompozycji i struktur formalnych (hierarchii klas dekorowalnych i klas dekoratorów), podczas gdy domieszki to technika oparta na dziedziczeniu. Domieszki bazujące na typach parametryzowanych można traktować jako uogólniony mechanizm dekoracji który nie wymaga struktury dziedziczenia typowej dla wzorca projektowego Decorator.

Poprzedni przykład moglibyśmy przeprojektować pod kątem rzeczonego wzorca:

```
// : generics/decorator/Decoration.java
package generics.decorator;
import java.util.*;

class Basic {
  private String value;
  public void set(String val) { value = val; }
  public String get() { return value; }
}

class Decorator extends Basic {
  protected Basic basic;
  public Decorator(Basic basic) { this.basic = basic; }
  public void set(String val) { basic.set(val); }
  public String get() { return basic.get(); }
}

class TimeStamped extends Decorator {
  private final long timeStamp;
  public TimeStamped(Basic basic) {
    super(basic);
    timeStamp = new Date().getTime();
  }
  public long getStamp() { return timeStamp; }
}

class SerialNumbered extends Decorator {
  private static long counter = 1;
  private final long serialNumber = counter++;
  public SerialNumbered(Basic basic) { super(basic); }
  public long getSerialNumber() { return serialNumber; }
}

public class Decoration {
  public static void main(String[] args) {
    TimeStamped t = new TimeStamped(new Basic());
    TimeStamped t2 = new TimeStamped(
      new SerialNumbered(new Basic()));
```

```
    //! t2.getSerialNumber(); // Niedostępne
    SerialNumbered s = new SerialNumbered(new Basic());
    SerialNumbered s2 = new SerialNumbered(
      new TimeStamped(new Basic()));
    //! s2.getStamp(); // Niedostępne
  }
} ///:~
```

Klasa powstająca z domieszkowania zawiera komplet interesujących nas metod, ale typ obiektu powstającego przy użyciu dekoratorów jest tożsamy z ostatnim typem dekorującym, więc choć *można* dodać więcej niż jedną warstwę, właściwy typ określa warstwa ostatnia, a zatem widoczne są tylko jej metody, podczas gdy typ klasy mieszanej obejmuje *wszystkie* typy domieszek łącznie. Zasadniczą wadą wzorca projektowego Decorator jest więc to, że działa on efektywnie jedynie z jedną warstwą dekoracji (tą ostatnią), zaś za domieszkami przemawia niewątpliwa naturalność. Wzorzec projektowy Decorator jest zatem jedynie ograniczonym co do zakresu rozwiązaniem problemu rozwiązywanego przez domieszki.

Ćwiczenie 38. Utwórz prosty system dekoracji, zaczynając od podstawowego typu reprezentującego czarną kawę, a następnie uzupełniając go o dekoratory: śmietankę, cukier, piankę, czekoladę, karmel i rum (4).

Domieszki w postaci dynamicznych proxy

Za pomocą dynamicznego proxy (działanie dynamicznych proxy opisuje rozdział „Informacje o typach") można modelować mechanizm domieszek znacznie wierniej niż przy użyciu wzorca projektowego Decorator. Przy zastosowaniu dynamicznego proxy *dynamiczny* typ wynikowej klasy to połączenie wszystkich domieszkowanych typów.

Z racji ograniczeń dynamicznych proxy każda domieszkowana klasa musi stanowić implementację ustalonego interfejsu:

```
//: generics/DynamicProxyMixin.java
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Tuple.*;

class MixinProxy implements InvocationHandler {
  Map<String,Object> delegatesByMethod;
  public MixinProxy(TwoTuple<Object,Class<?>>... pairs) {
    delegatesByMethod = new HashMap<String,Object>();
    for(TwoTuple<Object,Class<?>> pair : pairs) {
      for(Method method : pair.second.getMethods()) {
        String methodName = method.getName();
        // Pierwszy interfejs odwzorowania
        // implementuje metodę.
        if (!delegatesByMethod.containsKey(methodName))
          delegatesByMethod.put(methodName, pair.first);
      }
    }
  }
  public Object invoke(Object proxy, Method method,
    Object[] args) throws Throwable {
    String methodName = method.getName();
```

```
    Object delegate = delegatesByMethod.get(methodName);
    return method.invoke(delegate, args);
  }
  @SuppressWarnings("unchecked")
  public static Object newInstance(TwoTuple... pairs) {
    Class[] interfaces = new Class[pairs.length];
    for(int i = 0; i < pairs.length; i++) {
      interfaces[i] = (Class)pairs[i].second;
    }
    ClassLoader cl =
      pairs[0].first.getClass().getClassLoader();
    return Proxy.newProxyInstance(
      cl, interfaces, new MixinProxy(pairs));
  }
}

public class DynamicProxyMixin {
  public static void main(String[] args) {
    Object mixin = MixinProxy.newInstance(
      tuple(new BasicImp(), Basic.class),
      tuple(new TimeStampedImp(), TimeStamped.class),
      tuple(new SerialNumberedImp(),SerialNumbered.class));
    Basic b = (Basic)mixin;
    TimeStamped t = (TimeStamped)mixin;
    SerialNumbered s = (SerialNumbered)mixin;
    b.set("Ahoj");
    System.out.println(b.get());
    System.out.println(t.getStamp());
    System.out.println(s.getSerialNumber());
  }
} /* Output: (Sample)
Ahoj
1132519137015
1
*///:~
```

Ponieważ komplet typów domieszki widoczny jest jedynie w typie dynamicznym, a nie statycznym, wciąż nie dotarliśmy do rozwiązania znanego z języka C++, bo w celu wywołania metody typu domieszkującego jesteśmy zmuszani do rzutowania typu mieszanego w dół, ale stąd do prawdziwych domieszek już całkiem niedaleko.

Obsługa domieszek w Javie była przedmiotem całkiem intensywnych prac, z nowym dodatkiem do języka na czele — mowa o języku Jam przystosowanym właśnie do tworzenia klas mieszanych.

Ćwiczenie 39. Dodaj do programu *DynamicProxyMixin.java* nową klasę domieszki `Colored`, domieszaj ją do typu `mixin` i pokaż, że domieszka działa (1).

Typowanie utajone

Od początku tego rozdziału zmagamy się z koncepcją pisania kodu na tyle ogólnego, aby dało się go stosować jak najszerzej. W tym celu musimy poluzować ograniczenia co do typów, z którymi nasz kod może działać i na których operuje, nie zamierzając rezygnować

z zalet statycznej kontroli typów. W takim układzie możemy pisać kod, który daje się zastosować bez zmian w większej liczbie kontekstów — czyli kod uogólniony.

Typy ogólne w Javie zdają się zmierzać we właściwym kierunku. Kiedy piszemy albo korzystamy z uogólnień służących jedynie do przechowywania obiektów, możemy ich używać z dowolnymi typami z wyjątkiem typów podstawowych (ale tę niemożność łagodzą mechanizmy automatycznego pakowania wartości podstawowych w odpowiednie obiekty). Innymi słowy, proste uogólnienia służące do przechowywania obiektów mogą śmiało deklarować niewrażliwość na typ obiektów, na których operują. Kod, który posiada taką cechę i z tego względu nadaje się do najszerszego możliwego stosowania, jest kodem prawdziwie ogólnym.

Przekonałeś się już, że pożądana ogólność łamie się dopiero przy próbach manipulowania na typach uogólnionych (jeśli mają to być operacje spoza zestawu metod klasy `Object`) — zacieranie wymaga wtedy określenia ram typów zdatnych do użycia w ramach typu uogólnionego. To poważne zmniejszenie „ogólności", bo ramy ograniczają zestaw typów zdatnych do używania w uogólnieniu do pewnej wspólnej hierarchii klas albo implementacji wspólnych interfejsów. W niektórych przypadkach uogólnienie traci wtedy całkowicie swoją elastyczność, bo równie dobrze można by po prostu ograniczyć typ do konkretnej klasy (i jej podklas) albo implementacji konkretnego interfejsu.

W niektórych językach programowania ograniczenie to obchodzi się za pośrednictwem techniki tak zwanego *utajonego typowania* (ang. *latent typing*) albo *typowania strukturalnego*. Niekiedy mówi się też o typowaniu „kaczkowym" (ang. *duck typing*): „jeśli coś chodzi jak kaczka i gada jak kaczka, można traktować to coś jak kaczkę".

Kod uogólniony ogranicza się zwykle do kilku wywołań na rzecz uogólnionego typu, a w językach z typowaniem utajonym ograniczenia co do tego typu są rozluźniane przez wymóg implementacji zestawu wykorzystywanych metod, bez wymogu klasyfikacji typu do konkretnej klasy czy interfejsu. Takie typowanie pozwala na uogólnianie wskroś rozłącznych hierarchii klas i wywoływanie metod spoza wspólnego interfejsu. Kod może więc wyrażać takie oczekiwanie co do obrabianego typu: „nie interesuje mnie twój konkretny typ; ważne, żebyś znał metody `dajGłos()` i `siad()`". Brak wymogu konkretnego typu czy interfejsu to większa ogólność kodu.

Typowanie utajone to mechanizm organizacji i ponownego używania kodu. Można dzięki niemu pisać fragmenty kodu, które nadają się do prostszego stosowania w rozmaitych kontekstach, a organizacja kodu i jego zdatność do ponownego użycia to podstawowe aspekty programowania w ogóle: zawsze i wszędzie chodzi przecież o to, aby raz napisany kod działał w wielu miejscach. Brak wymogu nazywania konkretnego interfejsu, na którym kod będzie operował, zwiększa zakres zastosowań kodu.

Przykładami języków obsługujących typowanie utajone są Python (do pobrania nieodpłatnie z witryny *www.Python.org*) i C++[6]. Python to język z typowaniem dynamicznym (całość kontroli typów odbywa się w czasie wykonania), a C++ to język typowany statycznie (kontrola typów odbywa się w czasie kompilacji) — jak widać, utajone typowanie nie jest związane z konkretnym modelem typowania (dynamicznego czy statycznego).

[6] Typowanie utajone obsługują również takie języki jak Ruby czy Smalltalk.

Jeśli weźmiemy niedawny opis i wcielimy go w kod w języku Python, otrzymamy coś takiego:

```
#: generics/DogsAndRobots.py

class Dog:
   def speak(self):
      print "Hau!"
   def sit(self):
      print "Siada"
   def reproduce(self):
      pass


class Robot:
   def speak(self):
      print "Klik!"
   def sit(self):
      print "Zgrrrt!"
   def oilChange(self):
      pass

def perform(anything):
   anything.speak()
   anything.sit()

a = Dog()
b = Robot()
perform(a)
perform(b)
#:~
```

W języku Python zasięg jest określany wcięciami kodu (nie ma więc potrzeby stosowania nawiasów klamrowych). Znak '#' oznacza początek komentarza — komentarz rozciąga się od tego znaku aż do końca wiersza (a wiec działa jak symbol '//' w Javie). Metody klas jawnie podają argument wywołania reprezentujący obiekt, na rzecz którego nastąpiło wywołanie, który tu nazywa się `self` (w Javie jest to `this`). Wywołania konstruktora nie wymagają opatrywania dodatkowo słowem kluczowym `new`. Język Python pozwala też na definiowanie metod zewnętrznych wobec klas, czyli funkcji; przykładem takiej funkcji jest tu `perform()`.

Parametr funkcji `peform(anything)` nie posiada określenia typu — `anything` jest zwyczajnym identyfikatorem. Sęk w tym, aby ten identyfikator reprezentował obiekt obsługujący wszystkie operacje, którym jest poddawany w ciele funkcji `perform()` — można więc mówić o niejawnym ustaleniu wymogu implementacji interfejsu. Ale interfejs ten nie został nigdzie nazwany — jest on *utajony*. Funkcja `perform()` nie troszczy się zupełnic o typ argumentu — można do niej przekazywać dowolne obiekty dopóty, dopóki będą one obsługiwały wywołania `speak()` i `sit()`. Jeśli do wywołania `perform()` przekażemy obiekt niespełniający wymaganego interfejsu, spowodujemy wyjątek czasu wykonania.

Ten sam efekt możemy uzyskać w języku C++:

```
//: generics/DogsAndRobots.cpp

class Dog {
public:
  void speak() {}
  void sit() {}
  void reproduce() {}
};

class Robot {
public:
  void speak() {}
  void sit() {}
  void oilChange() {
};

template<class T> void perform(T anything) {
  anything.speak();
  anything.sit();
}

int main() {
  Dog d;
  Robot r;
  perform(d);
  perform(r);
} ///:~
```

I w Pythonie i w C++ typy `Dog` i `Robot` nie mają ze sobą nic wspólnego poza tym, że posiadają metody o identycznych nazwach, a — ogólniej — sygnaturach. Z punktu widzenia systemu typów są to jednak zupełnie niezależne i niezwiązane klasy. Jednakże funkcja `perform()` nie zajmuje się w ogóle konkretnym typem argumentu wywołania, a typowanie utajone pozwala na przekazywanie w wywołaniu egzemplarzy obu klas.

Język C++ kontroluje faktyczną możliwość wysłania komunikatów do obu obiektów. W przypadku próby przekazania do funkcji obiektu niewłaściwego typu kompilator zgłosi błąd (sam komunikat o takim błędzie byłby rozwlekły i niejasny, co jest jedną z przyczyn złej reputacji szablonów w języku C++). I choć kontrola typów odbywa się w różnych momentach — w C++ statycznie, a więc na etapie kompilacji, a w Pythonie dynamicznie, czyli w czasie wykonania — oba języki uniemożliwiają wykorzystanie niewłaściwych typów, przez co oba są uznawane za języki ze *ścisłą kontrolą typów* (ang. *strongly typed*)[7]. Typowanie utajone nie unieważnia ani nie znosi ścisłej kontroli typów.

Ponieważ uogólnienia pojawiły się w języku Java stosunkowo późno, nie było żadnej możliwości implementacji typowania utajonego, więc w Javie nie możemy korzystać z jego niewątpliwych zalet. W efekcie mechanizm uogólnienia w Javie jawi się jako mniej

[7] Niektórzy podnoszą, że z racji możliwości rzutowania typów, pozwalającej na omijanie systemu kontroli typów, C++ można uznać najwyżej za język ze słabą kontrolą typów (ang. *weak typed*). Najbliższe prawdy byłoby pewnie stwierdzenie, że C++ jest językiem z silną kontrolą typów i furtką do jej omijania.

„ogólny" niż mechanizm typowania utajonego w innych językach programowania[8]. Gdybyśmy spróbowali zaimplementować przykład wałkowany w dwóch innych językach w Javie, musielibyśmy określić typ argumentu poprzez klasę albo interfejs wymienione w wyrażeniu ramującym parametr typowy uogólnienia:

```
//: generics/Performs.java

public interface Performs {
  void speak();
  void sit();
} ///:~

//: generics/DogsAndRobots.java
// Brak typowania utajonego w Javie
import typeinfo.pets.*;
import static net.mindview.util.Print.*;

class PerformingDog extends Dog implements Performs {
  public void speak() { print("Hau!"); }
  public void sit() { print("Siada"); }
  public void reproduce() {}
}

class Robot implements Performs {
  public void speak() { print("Klik!"); }
  public void sit() { print("Zgrrt!"); }
  public void oilChange() {}
}

class Communicate {
  public static <T extends Performs>
  void perform(T performer) {
    performer.speak();
    performer.sit();
  }
}

public class DogsAndRobots {
  public static void main(String[] args) {
    PerformingDog d = new PerformingDog();
    Robot r = new Robot();
    Communicate.perform(d);
    Communicate.perform(r);
  }
} /* Output:
Hau!
Siada
Klik!
Zgrrt!
*///:~
```

Zauważ jednak, że metoda `perform()` nie wymaga uogólniania. Mogłaby najzwyczajniej przyjmować obiekty implementujące `Performs`:

[8] Implementacja uogólnień w języku Java, bazująca na zacieraniu typów, jest niekiedy określana jako uogólnienie *drugiej kategorii*.

```
// : generics/SimpleDogsAndRobots.java
// Usuwanie uogólnienia; kod wciąż działa.

class CommunicateSimply {
  static void perform(Performs performer) {
    performer.speak();
    performer.sit();
  }
}

public class SimpleDogsAndRobots {
  public static void main(String[] args) {
    CommunicateSimply.perform(new PerformingDog());
    CommunicateSimply.perform(new Robot());
  }
} /* Output:
Hau!
Siada
Klik!
Zgrrt!
*///:~
```

W tym przypadku uogólnienie było najzwyczajniej niepotrzebne, bo obie klasy wykorzystywane w `perform()` i tak musiały implementować wspólny interfejs — `Performs`.

Kompensacja braku typowania utajonego

Choć Java nie obsługuje typowania utajonego, nie oznacza to wcale, że nie można stosować uogólnionego (choć ujętego w ramę) kodu dla typów niepołączonych hierarchiami dziedziczenia. Okazuje się więc, że można tworzyć kod prawdziwie ogólny, ale wymaga to pewnych dodatkowych nakładów.

Refleksja

Po pierwsze, można się uciec do refleksji. Oto metoda `perform()` w wersji korzystającej z utajonego typowania w tym wydaniu:

```
// : generics/LatentReflection.java
// Zastosowanie refleksji do uzyskania typowania utajonego.
import java.lang.reflect.*;
import static net.mindview.util.Print.*;

// Nie implementuje interfejsu Performs:
class Mime {
  public void walkAgainstTheWind() {}
  public void sit() { print("Udaje, że siada"); }
  public void pushInvisibleWalls() {}
  public String toString() { return "Mim"; }
}

// Nie implementuje interfejsu Performs:
class SmartDog {
  public void speak() { print("Hau!"); }
```

```
  public void sit() { print("Siada"); }
  public void reproduce() {}
}

class CommunicateReflectively {
  public static void perform(Object speaker) {
    Class<?> spkr = speaker.getClass();
    try {
      try {
        Method speak = spkr.getMethod("speak");
        speak.invoke(speaker);
      } catch(NoSuchMethodException e) {
        print(speaker + " nie potrafi dać głosu");
      }
      try {
        Method sit = spkr.getMethod("sit");
        sit.invoke(speaker);
      } catch(NoSuchMethodException e) {
        print(speaker + " nie potrafi usiąść");
      }
    } catch(Exception e) {
      throw new RuntimeException(speaker.toString(), e);
    }
  }
}

public class LatentReflection {
  public static void main(String[] args) {
    CommunicateReflectively.perform(new SmartDog());
    CommunicateReflectively.perform(new Robot());
    CommunicateReflectively.perform(new Mime());
  }
} /* Output:
Hau!
Siada
Klik!
Zgrrt!
Mim nie potrafi dać głosu
Udaje, że siada
*///:~
```

Klasy są tu zupełnie rozłączne i nie posiadają wspólnych klas bazowych (oczywiście poza klasą `Object`) ani nie implementują wspólnych interfejsów. Za pośrednictwem mechanizmu refleksji metoda `CommunicateReflectively.perform()` może dynamicznie ustalać, czy argument wywołania posiada pożądane metody i ewentualnie je wywoływać. W ten sposób metoda radzi sobie nawet z obiektem typu `Mime`, który ma tylko część potrzebnych metod i tylko częściowo może spełnić zadanie metody `perform()`.

Aplikowanie metody do sekwencji obiektów

Refleksja daje ciekawe możliwości, ale oddala całość kontroli typów do czasu wykonania, co nie zawsze jest pożądane. Zwykle tam, gdzie to możliwe, chciałoby się polegać na statycznej kontroli typów. Czy można pogodzić kontrolę typów w czasie kompilacji z typowaniem utajonym?

Przyjrzyjmy się problemowi na przykładzie. Załóżmy, że chcemy utworzyć metodę apply(), która będzie aplikować dowolną metodę do każdego obiektu sekwencji. Nie przydadzą nam się interfejsy. Chcemy zaaplikować pewną (dowolną) metodę do kolekcji obiektów, a interfejsy nie pozwalają na opisanie „dowolnej metody". Jak osiągnąć cel w Javie?

Pierwsze podejście może polegać na zastosowaniu refleksji; dzięki zmiennym listom argumentów w Javie SE5 okazuje się to całkiem eleganckim rozwiązaniem:

```
// generics/Apply.java
// {main: ApplyTest}
import java.lang.reflect.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Apply {
  public static <T, S extends Iterable<? extends T>>
  void apply(S seq, Method f, Object... args) {
    try {
      for(T t: seq)
        f.invoke(t, args);
    } catch(Exception e) {
      // Niepowodzenia oznaczają błędy programistyczne
      throw new RuntimeException(e);
    }
  }
}

class Shape {
  public void rotate() { print(this + " rotate"); }
  public void resize(int newSize) {
    print(this + " resize " + newSize);
  }
}

class Square extends Shape {}

class FilledList<T> extends ArrayList<T> {
  public FilledList(Class<? extends T> type, int size) {
    try {
      for(int i = 0; i < size; i++)
        // Zakłada dostępność konstruktora domyślnego:
        add(type.newInstance());
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
}

class ApplyTest {
  public static void main(String[] args) throws Exception {
    List<Shape> shapes = new ArrayList<Shape>();
    for(int i = 0; i < 10; i++)
      shapes.add(new Shape());
    Apply.apply(shapes, Shape.class.getMethod("rotate"));
    Apply.apply(shapes,
      Shape.class.getMethod("resize", int.class), 5);
```

```
    List<Square> squares = new ArrayList<Square>();
    for(int i = 0; i < 10; i++)
      squares.add(new Square());
    Apply.apply(squares, Shape.class.getMethod("rotate"));
    Apply.apply(squares,
      Shape.class.getMethod("resize", int.class), 5);

    Apply.apply(new FilledList<Shape>(Shape.class, 10),
      Shape.class.getMethod("rotate"));
    Apply.apply(new FilledList<Shape>(Square.class, 10),
      Shape.class.getMethod("rotate"));

    SimpleQueue<Shape> shapeQ = new SimpleQueue<Shape>();
    for(int i = 0; i < 5; i++) {
      shapeQ.add(new Shape());
      shapeQ.add(new Square());
    }
    Apply.apply(shapeQ, Shape.class.getMethod("rotate"));
  }
} /* (Execute to see output) *///:~
```

W klasie `Apply` mamy szczęście, bo Java ma wbudowany interfejs `Iterable` wykorzystywany w bibliotece kontenerów. Z tego względu metoda `apply()` może przyjmować w wywołaniu wszystko, co implementuje interfejs `Iterable`, a więc między innymi wszystkie klasy `Collection`, z klasą `List` włącznie. Ale może też przyjmować wszystko inne, o ile uczynimy to coś implementacją `Iterable` — choćby własną, prostą klasę `SimpleQueue` zdefiniowaną tu, a wykorzystywaną powyżej, w metodzie `main()`:

```
//: generics/SimpleQueue.java
// Własny kontener, który również jest wcieleniem Iterable
import java.util.*;

public class SimpleQueue<T> implements Iterable<T> {
  private LinkedList<T> storage = new LinkedList<T>();
  public void add(T t) { storage.offer(t); }
  public T get() { return storage.poll(); }
  public Iterator<T> iterator() {
    return storage.iterator();
  }
} ///:~
```

W programie *Apply.java* wyjątki są konwertowane na `RuntimeException`, bo nie mamy za bardzo możliwości wycofania się ze stanu wyjątkowego — wyjątki w tym przypadku reprezentują błędy programistyczne, które trzeba poprawić w kodzie, a nie w czasie wykonania.

Zauważ, że aby przystosować klasy `Apply` i `FilledList` do użytku we wszystkich pożądanych sytuacjach, musiałem uciec się do ramowań i symboli wieloznacznych w uogólnieniach. Możesz spróbować się ich pozbyć, a odkryjesz, że w niektórych układach `Apply` i `FiledList` nie zadziałają.

Klasa `FilledList` reprezentuje pewien kłopot. Otóż użyteczny typ musi posiadać domyślny (bezargumentowy) konstruktor. Java nie ma środków, aby sprawdzić jego obecność w czasie kompilacji, więc jego brak staje się problemem czasu wykonania. Typowym

półśrodkiem wymuszającym kontrolę czasu kompilacji jest zdefiniowanie interfejsu wytwórni posiadającego metodę generującą obiekty; wtedy klasa `FilledList` przyjmowałaby taki interfejs w miejsce „surowej wytwórni" reprezentowanej nazwą typu. Tyle że wtedy wszystkie klasy wykorzystywane z `FilledList` musiałyby implementować interfejs wytwórni. Niestety, większość klas powstaje bez wiedzy o takim interfejsie, więc go nie implementuje. Rozwiązaniem byłyby adaptery — ale o nich później.

Zaprezentowane podejście, wykorzystujące nazwę typu, jest rozsądnym kompromisem, przynajmniej jak na rozwiązanie opracowane w pierwszym podejściu. Dzięki temu kompromisowi stosowanie `FilledList` jest na tyle proste, że jest szansa, iż faktycznie dojdzie do tego wykorzystania — to, co zbyt skomplikowane, jest zwykle ignorowane. Kłopotem jest jedynie sygnalizacja błędów w czasie wykonania — trzeba zadbać o to, aby ewentualne błędy pojawiły się na jak najwcześniejszym etapie produkcji oprogramowania.

Technika wykorzystująca nazwy typów jest zalecana w literaturze dotyczącej języka Java, choćby w artykule *Generics in the Java Programming Language*[9] Gilada Bracha, gdzie pada stwierdzenie: „to idiom wykorzystywany powszechnie choćby w nowych interfejsach programistycznych do manipulowania adnotacjami". Osobiście dostrzegam jednak pewną niespójność odnośnie poziomu akceptacji tej techniki — niektórzy mocno preferują podejście wykorzystujące wytwórnię.

Ponadto niezależnie od elegancji rozwiązania zastanego w Javie nie sposób nie zauważyć, że zastosowanie refleksji (choć w ostatnich wydaniach Javy znacznie się ona poprawiła) może być wolniejsze niż implementacje bez refleksji, a to z racji przeniesienia znacznej ilości operacji do czasu wykonania. Nie powinno to jednak dyskwalifikować rozwiązania, przynajmniej w pierwszym podejściu do problemu (inaczej narazilibyśmy się na zarzut przedwczesnej optymalizacji).

Ćwiczenie 40. Dodaj metodę `speak()` do wszystkich klas zwierzaków z pakietu `typeinfo.pets`. Zmodyfikuj program *Apply.java* tak, aby wywoływał metodę `speak()` dla heterogenicznej kolekcji obiektów `Pet` (3).

Kiedy nie ma pod ręką odpowiedniego interfejsu

Poprzedni przykład korzystał z faktu, że interfejs `Iterable` jest interfejsem wbudowanym i implementowanym we wszystkich klasach kolekcji, co było nam bardzo na rękę. Ale w przypadku ogólnym nie zawsze jest taki komfort — nie zawsze potrzebny interfejs jest pod ręką. Co wtedy?

Na potrzeby przykładu uogólnijmy pomysł z klasy `FilledList` i utwórzmy sparametryzowaną metodę `fill()`, która będzie przyjmować sekwencję i wypełniać ją za pomocą generatora (`Generator`). Kiedy spróbujemy zapisać taki kod w języku Java, wpadniemy w kłopoty, bo nie mamy do dyspozycji wygodnego interfejsu `Addable`, który pełniłby rolę interfejsu `Iterable` z poprzedniego przykładu. Dlatego zamiast określać typ wyrażeniem „cokolwiek, dla czego da się wywołać metodę `add()`", musimy wyrazić go wprost:

[9] Zobacz listę źródeł umieszczoną pod koniec niniejszego rozdziału.

„podtyp Collection". Powstały tak kod nie jest wielce ogólny, bo jest ograniczony do implementacji Collection. Nikła ogólność kodu ujawni się natychmiast, kiedy spróbujemy użyć go z klasą nieimplementującą interfejsu Collection. Wygląda to tak:

```
//: generics/Fill.java
// Uogólnianie pomysłu na klasę FilledList
// {main: FillTest}
import java.util.*;

// Nie działa z "czymkolwiek, co posiada metodę add()". Nie mamy
// do dyspozycji interfejsu "Addable", więc ograniczamy ogólność
// kodu do typów Collection. W tym przypadku uogólnienia nie
// oznaczają ogólności.
public class Fill {
  public static <T> void fill(Collection<T> collection,
  Class<? extends T> classToken, int size) {
    for(int i = 0; i < size; i++)
      // Zakłada dostępność konstruktora domyślnego:
      try {
        collection.add(classToken.newInstance());
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
  }
}

class Contract {
  private static long counter = 0;
  private final long id = counter++;
  public String toString() {
    return getClass().getName() + " " + id;
  }
}

class TitleTransfer extends Contract {}

class FillTest {
  public static void main(String[] args) {
    List<Contract> contracts = new ArrayList<Contract>();
    Fill.fill(contracts, Contract.class, 3);
    Fill.fill(contracts, TitleTransfer.class, 2);
    for(Contract c: contracts)
      System.out.println(c);
    SimpleQueue<Contract> contractQueue =
      new SimpleQueue<Contract>();
    // Nie zadziała. Metoda fill() nie jest dość ogólna:
    // Fill.fill(contractQueue, Contract.class, 3);
  }
} /* Output:
Contract 0
Contract 1
Contract 2
TitleTransfer 3
TitleTransfer 4
*///:~
```

Tu właśnie przydałby się mechanizm parametryzacji z utajonym typowaniem, bo nie bylibyśmy wtedy zdani na łaskę i niełaskę projektanta danej biblioteki i nie musielibyśmy przepisywać własnego kodu pod kątem każdej nowej biblioteki, w której nie uwzględniono naszej specyficznej sytuacji. Kod byłby prawdziwie ogólny. W powyższym przypadku projektanci języka Java nie przewidzieli interfejsu `Addable` (nic ma się czemu dziwić), więc jesteśmy ograniczeni do klas hierarchii `Collection` — klasa `SimpleQueue`, mimo że posiada metodę `add()`, nie zadziała. O otrzymanym kodzie trudno powiedzieć z czystym sumieniem, że jest „ogólny". Typowanie utajone zupełnie zmieniłoby sytuację.

Symulowanie typowania utajonego za pomocą adapterów

Skoro uogólnienia w Javie nie mają możliwości typowania utajonego, potrzebujemy czegoś podobnego, co pozwoliłoby na pisanie kodu dającego się aplikować dla typów przekraczających granice klas (otrzymując pożądany kod „ogólny"). Czy da się to jakoś zrobić?

Co dałoby nam tu typowanie utajone? Pozwalałoby zapisać kod mówiący: „nie jest ważne, na jakim typie operuję, dopóki ten typ posiada potrzebne metody". Typowanie utajnione tworzy więc niejawny, *nienazwany interfejs*, obejmujący owe „potrzebne metody". Gdybyśmy więc taki niezbędny interfejs zmontowali ręcznie (skoro nie zrobi tego za nas Java), rozwiązalibyśmy nasz problem.

Pisanie kodu tworzącego potrzebny nam interfejs na podstawie interfejsu posiadanego to przykład zastosowania wzorca projektowego Adapter. Adaptery mogą adaptować istniejące klasy do wymaganych interfejsów, a adaptacja ta odbywa się przy stosunkowo niewielkich nakładach programistycznych. Poniższe rozwiązanie, wykorzystujące zdefiniowane wcześniej klasy hierarchii `Coffee`, ilustruje różne sposoby pisania adapterów:

```
//: generics/Fill2.java
// Zastosowanie adapterów do symulowania typowania utajonego.
// {main: Fill2Test}
import generics.coffee.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

interface Addable<T> { void add(T t); }

public class Fill2 {
  // Wersja z nazwą klasy (literałem .class):
  public static <T> void fill(Addable<T> addable,
  Class<? extends T> classToken, int size) {
    for(int i = 0; i < size; i++)
      try {
        addable.add(classToken.newInstance());
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
  }
  // Wersja z generatorem:
  public static <T> void fill(Addable<T> addable,
  Generator<T> generator, int size) {
    for(int i = 0; i < size; i++)
```

```
      addable.add(generator.next());
    }
}

// Aby zaadaptować typ bazowy, należy skorzystać z kompozycji.
// Kompozycja pozwala na uczynienie każdej kolekcji wcieleniem Addable:
class AddableCollectionAdapter<T> implements Addable<T> {
  private Collection<T> c;
  public AddableCollectionAdapter(Collection<T> c) {
    this.c = c;
  }
  public void add(T item) { c.add(item); }
}

// Klasa pomocnicza do automatycznego wychwytywania typu:
class Adapter {
  public static <T>
  Addable<T> collectionAdapter(Collection<T> c) {
    return new AddableCollectionAdapter<T>(c);
  }
}

// Aby zaadaptować konkretny typ, należy użyć dziedziczenia.
// SimpleQueue będzie wcieleniem Addable na mocy dziedziczenia:
class AddableSimpleQueue<T>
extends SimpleQueue<T> implements Addable<T> {
  public void add(T item) { super.add(item); }
}

class Fill2Test {
  public static void main(String[] args) {
    // Adaptacja kolekcji Collection:
    List<Coffee> carrier = new ArrayList<Coffee>();
    Fill2.fill(
      new AddableCollectionAdapter<Coffee>(carrier),
      Coffee.class, 3);
    // Metoda pomocnicza wychwytująca typ:
    Fill2.fill(Adapter.collectionAdapter(carrier),
      Latte.class, 2);
    for(Coffee c: carrier)
      print(c);
    print("----------------------");
    // Użycie adaptowanej klasy:
    AddableSimpleQueue<Coffee> coffeeQueue =
      new AddableSimpleQueue<Coffee>();
    Fill2.fill(coffeeQueue, Mocha.class, 4);
    Fill2.fill(coffeeQueue, Latte.class, 1);
    for(Coffee c: coffeeQueue)
      print(c);
  }
} /* Output:
Coffee 0
Coffee 1
Coffee 2
Latte 3
Latte 4
```

```
---------------------
Mocha 5
Mocha 6
Mocha 7
Mocha 8
Latte 9
*///:~
```

Klasa `Fill2` nie wymaga typu `Collection`, jak to miało miejsce w przypadku klasy `Fill`. Wymaga jedynie, aby typ, na którym operuje, implementował interfejs `Addable`, który został napisany wyłącznie dla potrzeb klasy `Fill2` — to właśnie przejaw typowania utajonego, które kompilator miał wykonać za nas.

W tej wersji dodałem przeciążoną metodę `fill()`, która zamiast literału klasy (do tworzenia egzemplarza za pośrednictwem konstruktora domyślnego) przyjmuje generator. `Generator` jest bezpieczny ze względu na typ w czasie kompilacji — kompilator upewnia się co do przekazania odpowiedniego generatora, więc nie ma ryzyka zgłoszenia wyjątków.

Pierwszy adapter, w postaci klasy `AddableCollectionAdapter`, działa z typem bazowym `Collection`, co oznacza, że nadaje się do adaptowania dowolnych implementacji `Collection`. Ta wersja adaptera zachowuje referencję implementacji `Collection` i wykorzystuje ją do implementacji metody `add()`.

W przypadku adaptacji typu specjalizowanego (a nie bazowego typu hierarchii) można pisać nieco mniej kodu tworzącego adapter — wystarczy skorzystać z dziedziczenia, jak w `AddableSimpleQueue`.

W metodzie `Fill2Test.main()` można podziwiać poszczególne adaptery w działaniu. Na pierwszy ogień idzie kolekcja `Collection`, adaptowana przy użyciu adaptera `AddableCollectionAdapter`. Druga wersja tej operacji wykorzystuje uogólnioną metodę pomocniczą, która samodzielnie przechwytuje typ kolekcji, tak że nie trzeba go zapisywać jawnie — to wygodna sztuczka zwiększająca elegancję kodu.

Następny w kolejce jest adapter `AddableSimpleQueue`. W obu przypadkach adaptery pozwalają wykorzystywać w metodzie `Fill2.fill()` również klasy niemające nic wspólnego z interfejsem `Addable`.

Takie zastosowanie adapterów rekompensuje brak typowania utajonego, pozwalając na pisanie kodu prawdziwie ogólnego. Ale adaptacja to mimo wszystko dodatkowy etap programowania, którego konieczność musi być rozpoznawana zarówno przez twórcę, jak i użytkownika biblioteki, co w przypadku mniej doświadczonych programistów może okazać się problematyczne. Zaletą typowania utajonego jest właśnie eliminacja tego dodatkowego etapu, co wydatnie ułatwia pisanie i stosowanie kodu uogólnionego.

Ćwiczenie 41. Zmień program *Fill2.java* tak, aby operował nie na klasach hierarchii `Coffee`, a na klasach pakietu `typeinfo.pets` (1).

Obiekty funkcyjne w roli strategii

W tym ostatnim przykładzie utworzymy kod prawdziwie ogólny, wykorzystujący model adaptacji omawiany w poprzednim punkcie. Przykład ten pierwotnie stanowił próbę utworzenia operacji sumowania sekwencji elementów (dowolnego typu obsługującego operację dodawania), ostatecznie ewoluował zaś do operacji ogólnej, w oparciu o styl programowania zwany *programowaniem funkcyjnym*.

Przyglądając się procesowi polegającemu na próbie dodawania obiektów, przekonasz się, że to przypadek operacji wspólnej dla wielu klas niepowiązanych dziedziczeniem, więc sama operacja nie jest reprezentowana w żadnej wspólnej klasie bazowej — niekiedy da się ją reprezentować operatorem '+', innym razem będzie miała postać wywołania metody o nazwie `add()` albo podobnej. Właśnie z takimi sytuacjami mamy do czynienia przy pisaniu kodu uogólnionego, kiedy chcemy mieć możliwość stosowania go do wielu różnych klas — zwłaszcza, jak tu, kiedy klasy te już istnieją i nie można pozwolić sobie na ich „poprawianie". Zresztą nawet gdybyśmy ograniczyli nasze ambicje do klasy `Number` i jej podtypów, to ta klasa nie posiada niczego, co przystosowywałoby ją do naszych planów.

Rozwiązaniem jest zastosowanie wzorca projektowego *Strategy* („strategia"), który elegancko izoluje „to, co się zmienia" wewnątrz *obiektu funkcyjnego*[10]. Obiekt funkcyjny to obiekt, który w pewnym zakresie zachowuje się jak funkcja — zazwyczaj posiada jedną interesującą nas metodę implementującą „wywołanie obiektu" (w językach programowania dopuszczających przeciążanie operatorów w tej roli występuje często metoda przeciążająca operator wywołania funkcji). Zaletą obiektów funkcyjnych jest to, że w przeciwieństwie do zwyczajnych metod można je przekazywać pomiędzy fragmentami kodu, a ponadto obiekty funkcyjne mogą posiadać stan zachowywany pomiędzy wywołaniami. Oczywiście cechy te może przejawiać niemal dowolna klasa, dlatego wyróżnikiem obiektu funkcyjnego (jak w przypadku każdego wzorca projektowego) jest intencja jego utworzenia. Otóż naszą intencją jest utworzenie czegoś, co zachowuje się jak pojedyncza metoda, którą da się przekazywać; twór ten jest ściśle związany z wzorcem projektowym Strategy — a czasem nie daje się od niego odróżnić.

Jak w przypadku licznych innych wzorców projektowych dochodzi tu do niejakiego zacierania konturów: tworzymy obiekty funkcyjne realizujące adaptację, przekazywane do metod, gdzie mają pełnić rolę strategii.

Zgodnie z tym podejściem utworzyłem różne rodzaje metod uogólnionych, które pierwotnie musiałem utworzyć, i kilka innych. Oto efekt:

```
// : generics/Functional.java
import java.math.*;
import java.util.concurrent.atomic.*;
import java.util.*;
import static net.mindview.util.Print.*;
```

[10] Niekiedy *obiekty funkcyjne* określa się mianem *funktorów* — ja będę trzymał się terminu *obiekt funkcyjny*, bo *funktor* ma konkretne znaczenie w matematyce.

```
// Różne typy obiektów funkcyjnych:
interface Combiner<T> { T combine(T x, T y); }
interface UnaryFunction<R,T> { R function(T x); }
interface Collector<T> extends UnaryFunction<T,T> {
  T result(); // Wyłuskanie wyniku
}
interface UnaryPredicate<T> { boolean test(T x); }

public class Functional {
  // Wywołuje obiekt Combiner dla każdego elementu w
  // celu utworzenia wyniku zwracanego do wywołującego:
  public static <T> T
  reduce(Iterable<T> seq, Combiner<T> combiner) {
    Iterator<T> it = seq.iterator();
    if(it.hasNext()) {
      T result = it.next();
      while(it.hasNext())
        result = combiner.combine(result, it.next());
      return result;
    }
    // Jeśli sekwencja seq jest pusta
    return null; // Albo zgłosić wyjątek
  }
  // Przyjmuje obiekt funkcyjny i wywołuje go dla każdego obiektu
  // z listy, ignorując wartości zwracane kolejnych wywołań. Obiekt
  // funkcyjny może występować w roli parametru zbierającego i jako
  // taki jest na końcu zwracany do wywołującego.
  public static <T> Collector<T>
  forEach(Iterable<T> seq, Collector<T> func) {
    for(T t : seq)
      func.function(t);
    return func;
  }
  // Tworzy listę wyników przez wywołanie obiektu funkcyjnego
  // dla każdego obiektu z listy:
  public static <R,T> List<R>
  transform(Iterable<T> seq, UnaryFunction<R,T> func) {
    List<R> result = new ArrayList<R>();
    for(T t : seq)
      result.add(func.function(t));
    return result;
  }
  // Aplikuje unarny predykat dla każdego elementu sekwencji,
  // zwracając listę elementów, dla których predykat dał "true":
  public static <T> List<T>
  filter(Iterable<T> seq, UnaryPredicate<T> pred) {
    List<T> result = new ArrayList<T>();
    for(T t : seq)
      if(pred.test(t))
        result.add(t);
    return result;
  }
  // Aby użyć powyższych metod uogólnionych, musimy utworzyć
  // obiekty funkcyjne do adaptacji do poszczególnych potrzeb:
  static class IntegerAdder implements Combiner<Integer> {
    public Integer combine(Integer x, Integer y) {
      return x + y;
    }
```

```
}
static class
IntegerSubtracter implements Combiner<Integer> {
  public Integer combine(Integer x, Integer y) {
    return x - y;
  }
}
static class
BigDecimalAdder implements Combiner<BigDecimal> {
  public BigDecimal combine(BigDecimal x, BigDecimal y) {
    return x.add(y);
  }
}
static class
BigIntegerAdder implements Combiner<BigInteger> {
  public BigInteger combine(BigInteger x, BigInteger y) {
    return x.add(y);
  }
}
static class
AtomicLongAdder implements Combiner<AtomicLong> {
  public AtomicLong combine(AtomicLong x, AtomicLong y) {
    // Nie wiadomo, czy to ma jakieś znaczenie:
    return new AtomicLong(x.addAndGet(y.get()));
  }
}
static class BigDecimalUlp
implements UnaryFunction<BigDecimal,BigDecimal> {
  public BigDecimal function(BigDecimal x) {
    return x.ulp();
  }
}
static class GreaterThan<T extends Comparable<T>>
implements UnaryPredicate<T> {
  private T bound;
  public GreaterThan(T bound) { this.bound = bound; }
  public boolean test(T x) {
    return x.compareTo(bound) > 0;
  }
}
static class MultiplyingIntegerCollector
implements Collector<Integer> {
  private Integer val = 1;
  public Integer function(Integer x) {
    val *= x;
    return val;
  }
  public Integer result() { return val; }
}
public static void main(String[] args) {
  // Uogólnienia, zmienne listy argumentów i automatyczne
  // pakowanie w obiekty:
  List<Integer> li = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
  Integer result = reduce(li, new IntegerAdder());
  print(result);

  result = reduce(li, new IntegerSubtracter());
  print(result);
```

```
    print(filter(li, new GreaterThan<Integer>(4)));

    print(forEach(li,
      new MultiplyingIntegerCollector()).result());

    print(forEach(filter(li, new GreaterThan<Integer>(4)),
      new MultiplyingIntegerCollector()).result());

    MathContext mc = new MathContext(7);
    List<BigDecimal> lbd = Arrays.asList(
      new BigDecimal(1.1, mc), new BigDecimal(2.2, mc),
      new BigDecimal(3.3, mc), new BigDecimal(4.4, mc));
    BigDecimal rbd = reduce(lbd, new BigDecimalAdder());
    print(rbd);

    print(filter(lbd,
      new GreaterThan<BigDecimal>(new BigDecimal(3))));

    // Zastosowanie metod BigInteger
    // do generowania liczb pierwszych:
    List<BigInteger> lbi = new ArrayList<BigInteger>();
    BigInteger bi = BigInteger.valueOf(11);
    for(int i = 0; i < 11; i++) {
      lbi.add(bi);
      bi = bi.nextProbablePrime();
    }
    print(lbi);

    BigInteger rbi = reduce(lbi, new BigIntegerAdder());
    print(rbi);
    // Suma tej listy liczb pierwszych również jest liczbą pierwszą:
    print(rbi.isProbablePrime(5));

    List<AtomicLong> lal = Arrays.asList(
      new AtomicLong(11), new AtomicLong(47),
      new AtomicLong(74), new AtomicLong(133));
    AtomicLong ral = reduce(lal, new AtomicLongAdder());
    print(ral);

    print(transform(lbd,new BigDecimalUlp()));
  }
} /* Output:
28
-26
[5, 6, 7]
5040
210
11.000000
[3.300000, 4.400000]
[11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
311
true
265
[0.000001, 0.000001, 0.000001, 0.000001]
*///:~
```

Zacząłem od zdefiniowania interfejsów dla różnych typów obiektów funkcyjnych. Te zostały utworzone na żądanie, w miarę jak opracowywałem różne metody i odkrywałem potrzebę tworzenia kolejnych obiektów funkcyjnych. Klasa `Combiner` to propozycja anonimowego komentatora jednego z artykułów publikowanych na mojej stronie WWW. `Combiner` wyprowadza detale dodawania dwóch obiektów do osobnej abstrakcji, sygnalizując jedynie, że owe obiekty będą jakoś kombinowane ze sobą. W efekcie `IntegerAdder` i `IntegerSubtracter` mogą być podtypami klasy `Combiner`.

Interfejs `UnaryFunction` przyjmuje pojedynczy argument i zwraca wynik; argument i wynik nie muszą być tego samego typu. W roli „parametru zbierającego" występuje `Collector`, z którego po zakończeniu operacji można wyłuskać wynik zbiorczy. Z kolei `UnaryPredicate` zwraca wartość logiczną typu `boolean`. Możemy też definiować inne obiekty funkcyjne, ale już te wystarczą do zilustrowania omawianej koncepcji.

Klasa `Functional` zawiera zestaw metod uogólnionych aplikujących obiekty funkcyjne do sekwencji elementów. Metoda `reduce()` aplikuje funkcję obiektu funkcyjnego `Combiner` do każdego elementu sekwencji w celu zebrania wyniku ogólnego.

Metoda `forEach()` przyjmuje egzemplarz `Collector` i aplikuje jego funkcję do każdego elementu sekwencji, ignorując wyniki poszczególnych wywołań funkcji. Wywołania są podejmowane wyłącznie dla ich efektów ubocznych (co nie jest całkiem zgodne z duchem programowania funkcyjnego, ale może być użyteczne); ewentualnie sam `Collector` może wewnętrznie zachowywać stan modyfikowany kolejnymi wywołaniami i pełnić rolę parametru zbierającego.

Metoda `transform()` tworzy listę (sekwencję) elementów, wywołując funkcję obiektu funkcyjnego `UnaryFunction` dla każdego elementu listy roboczej; lista wynikowa to lista wyników wywołań.

Wreszcie metoda `filter()` aplikuje obiekt `UnaryFunction` do każdego obiektu przetwarzanej sekwencji i zachowuje na liście wynikowej te obiekty, dla których wywołanie zwróciło wartość `true`.

Idąc za ciosem, moglibyśmy definiować jeszcze inne funkcje uogólnione. Mnóstwo takich funkcji posiada standardowa biblioteka szablonów (STL) języka C++; tam funkcje owe noszą miano algorytmów. Znaczną liczbę funkcji uogólnionych można też znaleźć w licznych bibliotekach zewnętrznych, choćby w otwartej bibliotece JGA (Java Generic Algorithms).

W języku C++ dopasowanie typów do wywołań składane jest na barki typowania utajonego; w Javie trzeba napisać obiekty funkcyjne adaptujące metody uogólnione do konkretnych potrzeb, dlatego następna część klasy ilustruje różne implementacje obiektów funkcyjnych. Zwróć uwagę, że na przykład adaptery `IntegerAdder` i `BigDecimalAdder` rozwiązują identyczny problem — dodawania dwóch obiektów — przez wywołanie operacji odpowiednich dla adaptowanych typów. Mamy tu więc do czynienia z połączeniem wzorców projektowych *Adapter* i *Strategy*.

W metodzie `main()` można obserwować stosowanie poszczególnych obiektów funkcyjnych: każde wywołanie metody uogólnionej otrzymuje sekwencję elementów przetwarzanych wraz z odpowiednim obiektem funkcyjnym. Niektóre wyrażenia aplikujące obiekty funkcyjne mogą się mocno skomplikować, jak tu:

```
forEach(filter(li, new GreaterThan<Integer>(4)),
        new MultiplyingIntegerCollector()).result()
```

Takie wywołanie tworzy listę powstałą z wybrania wszystkich elementów listy roboczej `li` o wartości większej od 4; elementy tak powstałej listy są następnie kolejno poddawane działaniu obiektu funkcyjnego `MultiplyingIntegerCollector`; wynik zebrany w obiekcie funkcyjnym jest na końcu wyłuskiwany z niego wywołaniem metody `result()`. Szczegóły realizacji tej dość jednak skomplikowanej operacji najlepiej rozpoznać samemu, analizując poszczególne obiekty funkcyjne i wywołania metod uogólnionych.

Ćwiczenie 42. Utwórz dwie osobne klasy, niepowiązane ani dziedziczeniem, ani interfejsami. Obiekty obu klas powinny przechowywać wartości, a same klasy powinny posiadać przynajmniej metodę zwracającą tę wartość i metodę dokonującą modyfikacji wartości obiektu. Zmień program *Functional.java* tak, aby realizował operacje funkcyjne na kolekcjach obiektów obu nowych klas (operacje te nie muszą mieć charakteru arytmetycznego, jak w pierwotnej wersji *Functional.java*) (5).

Podsumowanie — czy rzutowanie jest aż tak złe?

Jako osoba zajmująca się wykładaniem tajników szablonów języka C++ od czasu, kiedy pojawiły się na scenie, jestem chyba częściej niż inni zmuszany do udzielania odpowiedzi na wątpliwość, o której za chwilę. I dopiero ostatnio musiałem się dłużej zastanowić nad tym, jak często okazuje się ona problematyczna — rozważając, jak często problem, który zamierzam opisać, faktycznie daje się we znaki programistom?

A chodzi o rzecz następującą. W rozdziale „Kolekcje obiektów" przekonałeś się, a w rozdziale „Kontenery z bliska" jeszcze przekonasz, że uogólnienia najlepiej nadają się do stosowania w klasach kontenerowych, takich jak `List`, `Set` czy `Map`. Przed pojawieniem się wersji Javy SE5 wszystko, co trafiało do takich kontenerów, musiało być rzutowane w górę na typ `Object`, co powodowało utratę informacji o typie przechowywanych elementów. Z kolei przy wyjmowaniu należało taki element rzutować na jego właściwy typ. Moim sztandarowym przykładem był kontener-lista (`List`) obiektów `Cat` (to odmiana przykładu z jabłkami i pomarańczami wykorzystanego na początku rozdziału „Kolekcje obiektów"). Bez wsparcia ze strony uogólnień dostępnych w Javie SE5 w kontenerach można było jedynie umieszczać obiekty klasy `Object` i takież obiekty z nich wyciągać. Nic nie zabezpieczało programu przed pojawieniem się obiektu `Dog` w kontenerze obiektów `Cat`.

Ale nawet bez uogólnień nie można było pomylić obiektów przechowywanych w kontenerach. Jeśli w kontenerze obiektów `Cat` wylądował `Dog`, to próba potraktowania wszystkich elementów kontenera jako obiektów `Cat` spowodowałaby w czasie wykonania wyjątek `RuntimeException` — wyjątek byłby reakcją na wyjęcie z kontenera referencji `Dog` i próbę rzutowania jej na typ `Cat`. Problem niezgodności typów nie pozostałby więc niezauważony, tyle że byłby wykryty dopiero w czasie wykonania, a nie już w czasie kompilacji.

W poprzednim wydaniu książki posunąłem się do stwierdzenia:

> *Ta sytuacja jest nie tylko irytująca. Może spowodować trudne do znalezienia błędy. Jeśli jeden fragment (lub kilka) programu umieszcza obiekty w kontenerze i odkryjemy w odrębnej części programu dzięki zgłoszeniu wyjątku, że dodano zły obiekt, trzeba znaleźć miejsce niewłaściwego wstawienia.*

Jednakże przy kolejnej okazji zacząłem się nad tym ponownie zastanawiać. Przede wszystkim, jak często do tego dochodzi? Nie pamiętam, żeby kiedykolwiek przytrafiło się to mnie osobiście, a i pytając różnych ludzi spotykanych na konferencjach o takie przypadki, nie usłyszałem ani jednego potwierdzenia. W jednej z książek widziałem przykład listy o nazwie `files` zawierającej obiekty klasy `String` (nazwy plików) — w tamtym przykładzie chodzi o ryzyko wstawienia do listy obiektu `File` (z powodu nazwy listy); może ryzyka tego można byłoby uniknąć, zmieniając najzwyczajniej nazwę listy na `fileNames`. Nawet najściślejsza kontrola typów w Javie nie powstrzyma krnąbrnego czy roztargnionego programisty od pisania niejasnego i mylącego kodu, a zły program, nawet jeśli się skompiluje, pozostaje wciąż złym programem. Zapewne większość programistów wybiera dla kontenerów nazwy sugerujące ich zawartość — w naszym przypadku kontener obiektów `Cat` nosiłby nazwę `cats` i już sama ta nazwa powinna wzbudzić niepokój u programisty, który zamierzałby wstawić do kontenera coś innego niż obiekt `Cat`. A nawet gdyby do tego doszło, to jak długo taka pomyłka pozostałaby niezauważona? Zdaje się, że przetrwałaby najwyżej do pierwszego poważniejszego testu operującego na próbkach prawdziwych danych.

U jednego z autorów spotkałem się ze stwierdzeniem, że taki błąd mógłby pozostać niezauważony „przez całe lata", ale jakoś umknął mi zalew doniesień o osobach odnajdujących „psy na listach kotów" w swoich aplikacjach. W rozdziale „Współbieżność" przekonasz się co prawda, że pewne kategorie błędów, zwłaszcza związane z wielowątkowością, mogą się ujawniać niezwykle rzadko, a i wtedy nie ujawniać wprost właściwego źródła problemu. Zapytuję więc, czy naprawdę problem „psa na liście kotów" miałby być uzasadnieniem do podejmowania wysiłku uzupełniania Javy o — jakby nie było — skomplikowany mechanizm omawiany w niniejszym rozdziale?

Osobiście uważam, że twórcom funkcji języka programowania, noszącej miano „uogólnień" (niekoniecznie w wydaniu charakterystycznym dla języka Java), przyświeca chęć zwiększenia *siły wyrazu* kodu, niekoniecznie zaś tworzenia bezpiecznych i kontrolowanych co do typu kontenerów. Te ostatnie są jakby typowym efektem ubocznym możliwości tworzenia kodu uogólnionego.

Nawet jeśli do uzasadniania uogólnień wykorzystuje się argument „psa wśród kotów", to uważam to za argument wątpliwy. I podtrzymuję to, co napisałem na początku rozdziału — że nie uważam, aby właśnie o to chodziło w tych całych uogólnieniach. Uogólnienia, zgodnie z ich nazwą, są środkiem zapisywania kodu bardziej ogólnego, to znaczy mniej ograniczonego co do typu, na którym operuje, a więc dającego się stosować w większej liczbie kontekstów. Lektura rozdziału powinna przekonać, że napisanie prawdziwie ogólnej klasy przechowującej (uproszczonego kontenera Javy) jest nieskomplikowane, ale już pisanie ogólnego kodu manipulującego na ogólnych typach wymaga dodatkowych nakładów, ponoszonych zarówno przez twórcę klasy, jak i przez ich użytkowników — obie strony muszą rozumieć i wdrażać wzorzec projektowy Adapter. Te dodatkowe nakłady zmniejszają niestety użyteczność uogólnień również tam, gdzie dawałyby istotne korzyści.

Do tego uogólnienia, jako mechanizm nieprzewidziany pierwotnie, a jedynie stanowiący przybudówkę do istniejącego i działającego z powodzeniem języka, uniemożliwiają zapewnienie kontenerom uogólnionym takiej solidności, jakiej można by od nich oczekiwać. Weźmy na przykład kontener `Map`, a szczególnie jego metody `containsKey(Object key)` i `get(Object key)`. Gdyby ta klasa była projektowana od początku pod kątem uogólnień, które istniałyby jako pierwotny i nieodłączny element języka, argumentami metod nie byłyby obiekty typu `Object`, a typu parametryzującego uogólnienie — to z kolei pozwoliłoby na realizację kontroli typów w czasie kompilacji, tak oczekiwanej ze strony uogólnień. W odpowiednikach takich kontenerów w języku C++ (`map`) typ klucza jest zawsze sprawdzany statycznie — w czasie kompilacji.

Jedno widać wyraźnie, wprowadzanie mechanizmu w rodzaju uogólnień do języka już dojrzałego i znajdującego się w powszechnym użyciu, stanowiącego podstawę niezliczonych instalacji, to bardzo odważna decyzja, której nie da się wdrożyć bezboleśnie. W języku C++ szablony (implementujące tam typy ogólne) zostały wprowadzone wraz z pierwszą wersją standardu ISO języka (a i tak nie obyło się bez kłopotów, bo przed pojawieniem się pierwszej wersji standardu C++ w użyciu były już wersje języka pozbawione obsługi szablonów), więc szablony były częścią języka niejako *od zawsze*. W Javie uogólnienia pojawiły się dopiero po niemal dziesięciu latach od momentu, jak sam język ujrzał światło dzienne, nie można więc było zignorować kwestii zgodności migracji, co z kolei musiało wywrzeć istotny wpływ na decyzje co do implementacji uogólnień. W efekcie programiści cierpią za grzech krótkowzroczności architektów Javy, którą ci przejawili, powołując do życia wersję 1.0 języka. Architekci ci musieli znać już szablony języka C++ i nawet rozważali wprowadzenie ich do języka, ale z jakichś względów zarzucili ten pomysł (są podstawy, by sądzić, że było to wynikiem pośpiechu). Ostatecznie ucierpiał i język, i jego użytkownicy. I tylko czas pokaże, jakie miejsce w języku zajmą uogólnienia i jaki będą miały wpływ na jego rozwój.

Są też języki, w których wcielono bardziej prostolinijną implementację typów parametryzowanych — mowa choćby o językach Nice (zobacz *http://nice.sourceforge.net*; język ten generuje kod bajtowy dla maszyny wirtualnej Javy i działa z istniejącymi bibliotekami Javy) i NextGen (*http://japan.cs.rice.edu/nextgen*). Trudno sobie jednak wyobrazić, aby któryś z nich stał się następcą Javy.

Dalsza lektura

- Artykuł autorstwa Gilada Brachy poświęcony uogólnieniom, zatytułowany *Generics in the Java Programming Language*, publikowany pod adresem *http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf*.
- Dokument *Java Generics FAQ* Angeliki Langer, bardzo przydatne źródło informacji, do wglądu pod adresem *http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html*.
- O symbolach wieloznacznych w typach uogólnionych można poczytać w publikacji zatytułowanej *Adding Wildcards to the Java Programming Language* — to praca zbiorowa Torgersona, Ernsta, Hansena, von den Ahe, Brachy i Gaftera, dostępna pod adresem *http://www.jot.fm/issues/issue_2004_12/article5*.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 16.

Tablice

Pod koniec rozdziału „Inicjalizacja i sprzątanie" dowiedziałeś się, jak definiować i inicjalizować tablice.

W najprostszym ujęciu tablice są tworem, który się konstruuje i wypełnia elementami, a potem można z tego tworu wybierać poszczególne elementy na podstawie indeksów mających postać wartości typu `int`; twory te, raz powołane do życia, nie zmieniają swoich rozmiarów. Zwykle taka wiedza zupełnie wystarcza do operowania tablicami, ale niekiedy stajemy przed zadaniem poddania tablicy bardziej wyszukanemu przetwarzaniu; warto też umieć szacować przydatność tablicy w porównaniu z którymś z bardziej przecież elastycznych kontenerów. Dlatego w tym rozdziale zajmiemy się właśnie tablicami.

Co w nich takiego specjalnego?

Tyle jest sposobów przechowywania obiektów, cóż więc specjalnego tkwi w tablicy?

Tablice odróżniają od innych rodzajów kontenerów dwie kwestie: wydajność i typ. Tablica jest bowiem w Javie najbardziej wydajnym sposobem zapisu i swobodnego dostępu do sekwencji obiektów (a dokładniej — do referencji do obiektów). Tablica jest prostą sekwencją liniową, która pozwala na szybki dostęp do elementów, ale trzeba za to zapłacić: kiedy tworzymy tablicę, jej rozmiar jest ustalony i nie może zostać zmieniony w czasie życia tablicy. Można by zaproponować stworzenie tablicy o konkretnym rozmiarze, a potem, kiedy zabraknie w niej już miejsca, stworzyć nową i przenieść wszystkie referencje z tej pierwszej do drugiej. Tak właśnie zachowuje się kontener `ArrayList`, ale z racji narzutu, koniecznego do uzyskania jego elastyczności, jego efektywność w porównaniu ze zwykłą tablicą wypada blado.

Ani tablicy, ani kontenera nic da się użyć niewłaściwie. Wykroczenie poza granicę czy to tablicy, czy to kontenera spowoduje zawsze zgłoszenie wyjątku `RuntimeException`.

Przed pojawieniem się w Javie typów ogólnych, klasy kontenerów traktowały przechowywane w nich obiekty tak, jakby nic posiadały określonego typu. To znaczy traktowały je jak egzemplarze typu `Object` — głównej klasy bazowej dla wszystkich klas w Javie. Wyższość tablic wobec kontenerów (ale sprzed ery typów ogólnych) przejawia się też w tym, że tablicę tworzymy w celu przechowywania obiektów jakiegoś określonego typu. Oznacza to,

że wystąpi błąd kompilacji, jeżeli spróbujemy zamieścić niepoprawny typ lub pomylimy się w typie, pobierając obiekt. Oczywiście Java powstrzyma nas przed wywołaniem niewłaściwej funkcji obiektu: albo podczas kompilacji, albo w czasie wykonania. Zatem żaden ze sposobów nie jest bardziej ryzykowny, ale lepiej jeśli to kompilator informuje — zmniejsza się wtedy prawdopodobieństwo, że końcowy użytkownik zostanie zaskoczony pojawieniem się wyjątku.

Tablica może przechowywać wartości typów podstawowych, do czego kontenery nieuogólnione nie były zdolne. Dzięki typom ogólnym kontenery mogą już określać i sprawdzać typy przechowywanych obiektów, a dzięki mechanizmowi pakowania w obiekty (ang. *autoboxing*) nie ma też problemu z przechowywaniem w kontenerach wartości typów podstawowych — konwersja na postać obiektów następuje automatycznie. Oto przykład porównujący tablice z kontenerami wzbogaconymi o typy ogólne:

```
//: arrays/ContainerComparison.java
import java.util.*;
import static net.mindview.util.Print.*;

class BerylliumSphere {
  private static long counter;
  private final long id = counter++;
  public String toString() { return "Sfera " + id; }
}

public class ContainerComparison {
  public static void main(String[] args) {
    BerylliumSphere[] spheres = new BerylliumSphere[10];
    for(int i = 0; i < 5; i++)
      spheres[i] = new BerylliumSphere();
    print(Arrays.toString(spheres));
    print(spheres[4]);

    List<BerylliumSphere> sphereList =
      new ArrayList<BerylliumSphere>();
    for(int i = 0; i < 5; i++)
      sphereList.add(new BerylliumSphere());
    print(sphereList);
    print(sphereList.get(4));

    int[] integers = { 0, 1, 2, 3, 4, 5 };
    print(Arrays.toString(integers));
    print(integers[4]);

    List<Integer> intList = new ArrayList<Integer>(
      Arrays.asList(0, 1, 2, 3, 4, 5));
    intList.add(97);
    print(intList);
    print(intList.get(4));
  }
} /* Output:
[Sfera 0, Sfera 1, Sfera 2, Sfera 3, Sfera 4, null, null, null, null, null]
Sfera 4
[Sfera 5, Sfera 6, Sfera 7, Sfera 8, Sfera 9]
Sfera 9
[0, 1, 2, 3, 4, 5]
4
[0, 1, 2, 3, 4, 5, 97]
4
*///:~
```

Oba modele przechowywania obiektów podlegają kontroli typów, a jedyną zewnętrzną różnicą jest to, że przy tablicach w odwołaniach do elementów stosuje się indeksowanie ([]), a na elementach kontenerów `List` operuje się metodami `add()` i `get()`. Podobieństwo tablic do kontenera `ArrayList` jest celowe, aby łatwo było się koncepcyjnie przestawiać z jednego modelu na drugi. Ale — o czym mówiliśmy w rozdziale „Kolekcje obiektów" — kontenery oferują zdecydowanie więcej niż tablice.

Wraz z pojawieniem się mechanizmu pakowania wartości podstawowych w obiekty kontenerów używa się z typami podstawowymi niemal równie łatwo jak tablic. Przy rozwiązywaniu ogólniejszych problemów ograniczenia tablic mogą jednak nieco uwierać — w takich przypadkach lepiej zdać się na jedną z klas kontenerowych.

Tablice to pełnoprawne obiekty

Bez względu na to, jakiego typu tablicę zastosujemy, identyfikator tablicy jest w rzeczywistości odwołaniem do prawdziwego obiektu, który został stworzony na stercie. Obiekt ten przechowuje odwołania do innych obiektów i może zostać stworzony bądź pośrednio jako skutek zapisu inicjalizującego, bądź jawnie przez użycie operatora `new`. Częścią obiektu tablicy (jedynym polem lub metodą, która jest dostępna z zewnątrz) jest składowa tylko do odczytu o nazwie `length`. Podaje ona, ile elementów może być zamieszczonych w tym obiekcie tablicy. Jedynym innym sposobem odwołania się do obiektu tablicowego jest zapis `[]`.

Poniższy przykład pokazuje różne sposoby inicjalizacji tablicy i przypisywania referencji do tablic różnym zmiennym tablicowym. Pokazuje on także, że tablice obiektów i tablice typów podstawowych są niemal identyczne w użyciu. Jedyna różnica polega na tym, że tablice obiektów przechowują referencje, podczas gdy tablice typów podstawowych bezpośrednio wartości.

```
// : arrays/ArrayOptions.java
// Inicjalizacja i przypisywanie tablic.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayOptions {
  public static void main(String[] args) {
    // Tablice obiektów:
    BerylliumSphere[] a; // Lokalna zmienna niezainicjalizowana
    BerylliumSphere[] b = new BerylliumSphere[5];
    // Referencje w tablicy są automatycznie
    // inicjalizowane wartościami pustymi (null):
    print("b: " + Arrays.toString(b));
    BerylliumSphere[] c = new BerylliumSphere[4];
    for(int i = 0; i < c.length; i++)
      if(c[i] == null) // test na obecność referencji pustej
        c[i] = new BerylliumSphere();
    // Inicjalizacja grupowa:
    BerylliumSphere[] d = { new BerylliumSphere(),
      new BerylliumSphere(), new BerylliumSphere()
    };
    // Dynamiczna inicjalizacja grupowa:
```

```
    a = new BerylliumSphere[]{
      new BerylliumSphere(), new BerylliumSphere(),
    };
    // (W obu przypadkach przecinek za ostatnim
    // inicjalizatorem jest opcjonalny)
    print("a.length = " + a.length);
    print("b.length = " + b.length);
    print("c.length = " + c.length);
    print("d.length = " + d.length);
    a = d;
    print("a.length = " + a.length);

    // Tablice wartości podstawowych:
    int[] e; // Referencja pusta
    int[] f = new int[5];
    // Wartości podstawowe w tablicy są automatycznie
    // inicjalizowane zerem:
    print("f: " + Arrays.toString(f));
    int[] g = new int[4];
    for(int i = 0; i < g.length; i++)
      g[i] = i*i;
    int[] h = { 11, 47, 93 };
    // Błąd kompilacji: niezainicjalizowana zmienna e:
    //! print("e.length = " + e.length);
    print("f.length = " + f.length);
    print("g.length = " + g.length);
    print("h.length = " + h.length);
    e = h;
    print("e.length = " + e.length);
    e = new int[]{ 1, 2 };
    print("e.length = " + e.length);
  }
} /* Output:
b: [null, null, null, null, null]
a.length = 2
b.length = 5
c.length = 4
d.length = 3
a.length = 3
f: [0, 0, 0, 0, 0]
f.length = 5
g.length = 4
h.length = 3
e.length = 3
e.length = 2
*///:~
```

Tablica a jest inicjalizowana odwołaniem pustym (`null`), toteż kompilator nie zezwala na robienie z nią czegokolwiek, zanim nie zostanie odpowiednio ustawiona. Tablica b została zainicjalizowana odwołaniem do tablicy referencji typu `BerylliumSphere`, ale żaden obiekt klasy `BerylliumSphere` nie został w niej umieszczony. Mimo to można spytać, *jaki rozmiar* posiada tablica, ponieważ b wskazuje już na poprawny obiekt. Ma to jedną drobną wadę: nic ma możliwości rozpoznania, ile obiektów *jest* rzeczywiście w tablicy, gdyż `length` mówi jedynie, ile elementów *może* zostać w niej zamieszczonych — zatem podaje rozmiar obiektu tablicowego, a nie liczbę elementów, które w danej chwili zawiera. Kiedy tworzony jest obiekt tablicowy, jego odwołania są automatycznie ustawiane na wartość `null`, toteż można sprawdzić, czy określone pole tablicy zawiera

obiekt, poprzez sprawdzenie, czy nie jest `null`. Podobnie tablice typów podstawowych są automatycznie inicjalizowane wartością zero dla typów liczbowych i wartościami `(char)0` dla `char` i `false` dla `boolean`.

Przykład tablicy `c` pokazuje możliwość utworzenia obiektu tablicowego i przypisania obiektów `BerylliumSphere` do wszystkich komórek tablicy. Tablica `d` jest inicjalizowana grupowo, co powoduje stworzenie obiektu tablicy (pośrednio poprzez `new` na stercie, tak jak w przypadku tablicy `c`) *i* zainicjowanie go obiektami `BerylliumSphere` — wszystko to w jednym wyrażeniu.

Następna inicjalizacja może być nazwana dynamiczną inicjalizacją grupową. Sposób inicjalizacji użyty w przypadku tablicy `d` może zostać zastosowany tylko w miejscu deklaracji `d`, natomiast drugi pozwala na stworzenie i zainicjalizowanie tablicy w dowolnym miejscu programu. Na przykład przypuśćmy, że metoda o nazwie `hide()` pobiera tablicę obiektów klasy `BerylliumSphere`. Można by ją wywołać następująco:

```
hide(d);
```

ale można również dynamicznie stworzyć tablicę przekazywaną jako argument metody:

```
hide(new BerylliumSphere[] { new BerylliumSphere(),
  new BerylliumSphere() });
```

W pewnych sytuacjach taki nowy zapis jest znacznie wygodniejszy.

Wyrażenie

```
a = d;
```

pokazuje, jak można pobrać odwołanie związane z jedną tablicą i przypisać je innej tablicy — tak jak w przypadku wszystkich innych rodzajów referencji do obiektów. Teraz już zarówno `a`, jaki i `d` wskazują na ten sam obiekt tablicowy umieszczony na stercie.

Druga część przykładu *ArrayOptions.java* dowodzi, że tablice wartości podstawowych funkcjonują podobnie jak tablice obiektów — *z tą różnicą*, że tablice wartości podstawowych przechowują wartości zmiennych bezpośrednio.

Ćwiczenie 1. Napisz metodę, która przyjmie w wywołaniu tablicę obiektów `BerylliumSphere`. Wywołaj metodę, tworząc dynamicznie argument wywołania. Wykaż, że w takim przypadku zwykła inicjalizacja grupowa nie zadziała. Wykryj sytuacje, w których zwykła inicjalizacja grupowa tablicy działa, i te, w których dynamiczna inicjalizacja grupowa jest nadmiarowa (2).

Tablice w roli wartości zwracanych

Przypuśćmy, że piszemy metodę, która ma zwracać nie tylko jedną rzecz, lecz całą grupę. Języki takie jak C czy C++ utrudniają to zadanie, gdyż nie można zwyczajnie zwrócić tablicy, a tylko co najwyżej wskaźnik do niej. Pojawia się pewien problem, ponieważ kontrola życia tablicy jest co najmniej nieelegancka, co bez wątpienia prowadzi do wycieków pamięci.

W Javie po prostu zwraca się tablicę. Tu programista nigdy nie musi przyjmować na siebie odpowiedzialności za tą tablicę — będzie ona istnieć tak długo, jak długo będzie potrzebna. Kiedy nie będzie nam już potrzebna, odśmiecacz ją zlikwiduje.

Jako przykład rozważmy zwracanie tablicy ciągów znaków:

```
//: arrays/IceCream.java
// Zwracanie tablic z metod.
import java.util.*;

public class IceCream {
  private static Random rand = new Random(47);
  static final String[] FLAVORS = {
    "Czekoladowe", "Truskawkowe", "Karmel Waniliowy",
    "Miętowa Kostka", "Karmel Mokka-Migdałowy", "Rumowy Rodzynek",
    "Krem Pralinkowy", "Borówkowe Ciastko"
  };
  public static String[] flavorSet(int n) {
    if(n > FLAVORS.length)
      throw new IllegalArgumentException("Za dużo smaków");
    String[] results = new String[n];
    boolean[] picked = new boolean[FLAVORS.length];
    for(int i = 0; i < n; i++) {
      int t;
      do
        t = rand.nextInt(FLAVORS.length);
      while(picked[t]);
      results[i] = FLAVORS[t];
      picked[t] = true;
    }
    return results;
  }
  public static void main(String[] args) {
    for(int i = 0; i < 7; i++)
      System.out.println(Arrays.toString(flavorSet(3)));
  }
} /* Output:
[Rumowy Rodzynek, Miętowa Kostka, Karmel Mokka-Migdałowy]
[Czekoladowe, Truskawkowe, Karmel Mokka-Migdałowy]
[Truskawkowe, Miętowa Kostka, Karmel Mokka-Migdałowy]
[Rumowy Rodzynek, Karmel Waniliowy, Borówkowe Ciastko]
[Karmel Waniliowy, Czekoladowe, Karmel Mokka-Migdałowy]
[Krem Pralinkowy, Truskawkowe, Karmel Mokka-Migdałowy]
[Karmel Mokka-Migdałowy, Truskawkowe, Miętowa Kostka]
*///:~
```

W metodzie `flavorSet()` tworzona jest tablica obiektów `String` o nazwie `results`. Rozmiar tej tablicy wynosi `n` wyznaczone wartością argumentu przekazanego do metody. Dalej w sposób losowy z tablicy `FLAVORS` wybierane są jakieś rodzaje lodów i umieszczane w tablicy `flavors`, która ostatecznie jest zwracana z metody. Zwrot tablicy jest tym samym, co zwrot każdego innego obiektu — jest to przecież referencja. Nie jest tu istotne, że tablica była stworzona wewnątrz metody `flavorSet()` czy gdziekolwiek indziej. Odśmiecacz pamięci zadba o zniszczenie tablicy dopiero wtedy, kiedy nie będzie już używana.

Zauważ, że podczas wyboru metoda `flavorSet()` upewnia się, że losowy wybór nie nastąpił już wcześniej. Jest to przeprowadzone w pętli `do`, która kontynuuje wytwarzanie liczb losowych aż do odnalezienia wartości, której jeszcze nie ma w tablicy `picked`. (Oczywiście można było również przeprowadzić takie sprawdzenie poprzez porównanie ciągów tekstowych wybieranych do `results`.) Kiedy wybór liczby się powiedzie, to dodawany jest wpis i szukana kolejna wartość (`i` jest inkrementowane).

Na wyjściu programu można sprawdzić, czy rzeczywiście `flavorSet()` wybiera smaki w sposób losowy.

Ćwiczenie 2. Napisz metodę, która przyjmie argument typu `int` i zwróci tablicę o rozmiarze odpowiadającej wartości argumentu wypełnioną obiektami `BerylliumSphere` (1).

Tablice wielowymiarowe

Java pozwala na łatwe tworzenie tablic wielowymiarowych. W przypadku tablic wartości podstawowych każdy wektor (wiersz) tablicy wielowymiarowej należy ująć w nawiasy klamrowe:

```
//: arrays/MultidimensionalPrimitiveArray.java
// Tworzenie tablic wielowymiarowych.
import java.util.*;

public class MultidimensionalPrimitiveArray {
  public static void main(String[] args) {
    int[][] a = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    System.out.println(Arrays.deepToString(a));
  }
} /* Output:
[[1, 2, 3], [4, 5, 6]]
*///:~
```

Każda para zagnieżdżonych nawiasów klamrowych to osobny wymiar tablicy.

Przykład wykorzystuje metodę `Arrays.deepToString()` z Javy SE5, która zamienia wielowymiarową tablicę na ciągi `String` — efekt widać na wyjściu programu.

Tablicę można też przydzielać za pomocą operatora `new`. Tablicę trójwymiarową można więc utworzyć również tak:

```
//: arrays/ThreeDWithNew.java
import java.util.*;

public class ThreeDWithNew {
  public static void main(String[] args) {
    // 3-wymiarowa tablica o ustalonym rozmiarze:
    int[][][] a = new int[2][2][4];
    System.out.println(Arrays.deepToString(a));
  }
```

```
} /* Output:
[[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
*///:~
```

Jak widać, jeśli elementy tablicy wartości podstawowych nie zostaną zainicjalizowane jawnie, dojdzie do automatycznej inicjalizacji stosowną wartością. Elementy tablic obiektów są z kolei automatycznie inicjalizowane referencjami pustymi.

Każdy wiersz tablicy tworzącej macierz może mieć dowolny rozmiar (takie tablice noszą miano *tablic nierównych* — ang. *ragged array*):

```
//: arrays/RaggedArray.java
import java.util.*;

public class RaggedArray {
  public static void main(String[] args) {
    Random rand = new Random(47);
    // 3-wymiarowa tablica z wierszami o zmiennym rozmiarze:
    int[][][] a = new int[rand.nextInt(7)][][];
    for(int i = 0; i < a.length; i++) {
      a[i] = new int[rand.nextInt(5)][];
      for(int j = 0; j < a[i].length; j++)
        a[i][j] = new int[rand.nextInt(5)];
    }
    System.out.println(Arrays.deepToString(a));
  }
} /* Output:
[[], [[0], [0], [0, 0, 0, 0]], [[], [0, 0], [0, 0]], [[0, 0, 0], [0], [0, 0, 0, 0]], [[0, 0, 0], [0, 0, 0], [0], []], [[0],
[], [0]]]
*///:~
```

Pierwsze wywołanie new tworzy tablicę z losowo dobranym rozmiarem pierwszego wiersza i nieznanymi rozmiarami pozostałych. Drugie wywołanie new (w pętli) wypełnia drugi poziom, pozostawiając nieznaną wartość trzeciego indeksu — ta jest ustalana dopiero przy trzecim wywołaniu new.

Podobnie można operować tablicami obiektów (nie wartości podstawowych). Poniższy przykład pokazuje, jak zebrać wiele wyrażeń new w nawiasach klamrowych:

```
//: arrays/MultidimensionalObjectArrays.java
import java.util.*;

public class MultidimensionalObjectArrays {
  public static void main(String[] args) {
    BerylliumSphere[][] spheres = {
      { new BerylliumSphere(), new BerylliumSphere() },
      { new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere() },
      { new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere() },
    };
    System.out.println(Arrays.deepToString(spheres));
  }
} /* Output:
```

```
[[Sfera 0, Sfera 1], [Sfera 2, Sfera 3, Sfera 4, Sfera 5], [Sfera 6, Sfera 7, Sfera 8, Sfera 9, Sfera 10, Sfera
11, Sfera 12, Sfera 13]]
*///:~
```

Tablica spheres to kolejny przykład tablicy nierównej, w której każdy wiersz może mieć inny rozmiar.

W inicjalizacji tablicy można polegać na mechanizmie pakowania wartości podstawowych w obiekty:

```
//: arrays/AutoboxingArrays.java
import java.util.*;

public class AutoboxingArrays {
  public static void main(String[] args) {
    Integer[][] a = { // Autoboxing:
      { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },
      { 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 },
      { 51, 52, 53, 54, 55, 56, 57, 58, 59, 60 },
      { 71, 72, 73, 74, 75, 76, 77, 78, 79, 80 },
    };
    System.out.println(Arrays.deepToString(a));
  }
} /* Output:
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [21, 22, 23, 24, 25, 26, 27, 28, 29, 30], [51, 52, 53, 54, 55, 56, 57, 58, 59, 60],
[71, 72, 73, 74, 75, 76, 77, 78, 79, 80]]
*///:~
```

A oto przykład tworzenia tablicy obiektów (nie wartości podstawowych) kawałek po kawałku:

```
//: arrays/AssemblingMultidimensionalArrays.java
// Tworzenie tablic wielowymiarowych.
import java.util.*;

public class AssemblingMultidimensionalArrays {
  public static void main(String[] args) {
    Integer[][] a;
    a = new Integer[3][];
    for(int i = 0; i < a.length; i++) {
      a[i] = new Integer[3];
      for(int j = 0; j < a[i].length; j++)
        a[i][j] = i * j; // Pakowanie w obiekty
    }
    System.out.println(Arrays.deepToString(a));
  }
} /* Output:
[[0, 0, 0], [0, 1, 2], [0, 2, 4]]
*///:~
```

Wyrażenie `i*j` służy tu jedynie temu, aby tablica nie była wypełniana wciąż takimi samymi wartościami.

Metoda `Arrays.deepToString()` działa równie dobrze dla tablic wartości podstawowych, jak i tablic obiektów:

```
// arrays/MultiDimWrapperArray.java
// Wielowymiarowe tablice obiektów "opakowań".
import java.util.*;

public class MultiDimWrapperArray {
  public static void main(String[] args) {
    Integer[][] a1 = { // Autoboxing
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    Double[][][] a2 = { // Autoboxing
      { { 1.1, 2.2 }, { 3.3, 4.4 } },
      { { 5.5, 6.6 }, { 7.7, 8.8 } },
      { { 9.9, 1.2 }, { 2.3, 3.4 } },
    };
    String[][] a3 = {
      { "Pchnąć", "w", "tę", "łódź" },
      { "jeża", "lub" },
      { "ośm", "skrzyń", "fig", "łaskaw", "bądź", "druhu" },
    };
    System.out.println("a1: " + Arrays.deepToString(a1));
    System.out.println("a2: " + Arrays.deepToString(a2));
    System.out.println("a3: " + Arrays.deepToString(a3));
  }
} /* Output:
a1: [[1, 2, 3], [4, 5, 6]]
a2: [[[1.1, 2.2], [3.3, 4.4]], [[5.5, 6.6], [7.7, 8.8]], [[9.9, 1.2], [2.3, 3.4]]]
a3: [[Pchnąć, w, tę, łódź], [jeża, lub], [ośm, skrzyń, fig, łaskaw, bądź, druhu]]
*///:~
```

Znów przy wypełnianiu tablic obiektów typu `Integer` i `Double` Java mogła się wykazać mechanizmem automatycznego pakowania wartości podstawowych w stosowne obiekty.

Ćwiczenie 3. Napisz metodę, która utworzy i zainicjalizuje dwuwymiarową tablicę wartości `double`. Rozmiar tablicy ma być określany na podstawie argumentów wywołania metody, a wartości inicjalizujące mają być wybierane z zakresu wyznaczonego wartościami, początkową i końcową, które również mają być przekazywane do metody za pośrednictwem argumentów. Napisz drugą metodę, która wypisze tablicę wygenerowaną przez pierwszą metodę. W metodzie `main()` przetestuj obie metody, tworząc i wypisując kilka tablic o różnych rozmiarach (4).

Ćwiczenie 4. Powtórz poprzednie ćwiczenie dla tablicy trójwymiarowej (2).

Ćwiczenie 5. Wykaż, że tablice wielowymiarowe obiektów (wartości niepodstawowych) są automatycznie wypełniane referencjami pustymi (1).

Ćwiczenie 6. Napisz metodę, która przyjmie parę argumentów typu `int` reprezentujących rozmiary tablicy dwuwymiarowej. Metoda powinna zwrócić i wypełnić tablicę 2-wymiarową obiektów `BerylliumSphere` o odpowiednim rozmiarze (1).

Ćwiczenie 7. Powtórz poprzednie ćwiczenie dla tablicy trójwymiarowej (1).

Tablice a typy ogólne

Zasadniczo tablice i typy ogólne nie mają się zbytnio ku sobie. Nie można na przykład tworzyć tablic typów sparametryzowanych:

```
Peel<Banana>[] peels = new Peel<Banana>[10];   // niedozwolone
```

Mechanizm odwzorowania typów sparametryzowanych na typy niesparametryzowane (ang. *erasure*) powoduje utratę informacji o typie parametryzującym, a tablica musi znać dokładny typ elementów, inaczej nie mogłaby zapewnić kontroli typowania.

Można jednak sparametryzować typ samej tablicy:

```
//: arrays/ParameterizedArrayType.java

class ClassParameter<T> {
  public T[] f(T[] arg) { return arg; }
}

class MethodParameter {
  public static <T> T[] f(T[] arg) { return arg; }
}

public class ParameterizedArrayType {
  public static void main(String[] args) {
    Integer[] ints = { 1, 2, 3, 4, 5 };
    Double[] doubles = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Integer[] ints2 =
      new ClassParameter<Integer>().f(ints);
    Double[] doubles2 =
      new ClassParameter<Double>().f(doubles);
    ints2 = MethodParameter.f(ints);
    doubles2 = MethodParameter.f(doubles);
  }
} ///:~
```

Zwróć uwagę na wygodę stosowania sparametryzowanej metody zamiast parametryzowanej klasy: nie trzeba tworzyć egzemplarza klasy dla każdego z typów parametryzujących i można metodę uczynić statyczną. Oczywiście nie zawsze istnieje wybór pomiędzy sparametryzowaną metodą a klasą, ale tam, gdzie to możliwe, wypadałoby preferować metodę.

Jak się okazuje, stwierdzenie, że nie można tworzyć tablic typów ogólnych, jest nie do końca prawdziwe. Co prawda kompilator nie pozwoli na utworzenie egzemplarza tablicy typu ogólnego, ale umożliwi tworzenie referencji do takiej tablicy. Oto przykład:

```
List<String>[] ls;
```

Taki kod nie wzbudzi protestów kompilatora. I choć nie można utworzyć obiektu tablicy przechowującej uogólnienia, zawsze można utworzyć tablicę typu nieuogólnionego i dokonać rzutowania:

```
//: arrays/ArrayOfGenerics.java
// Można tworzyć tablice typów uogólnionych.
import java.util.*;
```

```
public class ArrayOfGenerics {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    List<String>[] ls;
    List[] la = new List[10];
    ls = (List<String>[])la; // Ostrzeżenie "Unchecked"
    ls[0] = new ArrayList<String>();
    // Kontrola czasu kompilacji spowoduje błąd:
    //! ls[1] = new ArrayList<Integer>();

    // Problem: List<String> jest podtypem typu Object
    Object[] objects = ls; // więc przypisanie jest uprawnione
    // To też kompiluje się i działa bez protestów:
    objects[1] = new ArrayList<Integer>();

    // Ale przy skromniejszych potrzebach można utworzyć
    // tablicę typu uogólnionego, tyle że powodując
    // ostrzeżenie "unchecked":
    List<BerylliumSphere>[] spheres =
      (List<BerylliumSphere>[])new List[10];
    for(int i = 0; i < spheres.length; i++)
      spheres[i] = new ArrayList<BerylliumSphere>();
  }
} ///:~
```

Po pozyskaniu referencji `List<String>[]` można uzyskać pewną kontrolę czasu kompilacji. Problem w tym, że tablice są kowariantne, więc `List<String>[]` to również `Object`, co można wykorzystać do przypisania do tablicy `ArrayList<Integer>` bez jakiegokolwiek błędu czy to w czasie kompilacji, czy w czasie wykonania.

Jeśli z góry wiesz, że nie będziesz wykonywał rzutowania w górę i Twoje potrzeby są stosunkowo skromne, możesz utworzyć tablicę typu ogólnego z elementarnymi opcjami kontroli w czasie kompilacji. Jednak w niemal każdym przypadku użycia takiej tablicy lepszy okaże się zapewne odpowiedni kontener.

Zasadniczo przekonasz się, że typy ogólne są efektywne na *granicach* klasy bądź metody. W obszarach wewnętrznych są zwykle bezużyteczne. Nie można na przykład utworzyć tablicy typu parametryzującego:

```
//: arrays/ArrayOfGenericType.java
// Tablice typu uogólnionego nie kompilują się.

public class ArrayOfGenericType<T> {
  T[] array; // OK
  @SuppressWarnings("unchecked")
  public ArrayOfGenericType(int size) {
    //! array = new T[size]; // niedozwolone
    array = (T[])new Object[size]; // Ostrzeżenie "unchecked"
  }
  // Niedozwolone:
  //! public <U> U[] makeArray() { return new U[10]; }
} ///:~
```

Mechanizm odwzorowania typów (erasure) znów wejdzie nam w drogę — próbujemy tu utworzyć tablice typów, które zostały już wymazane, i jako takie są nieznane kompilatorowi. Zauważ za to, że możesz utworzyć tablicę typu `Object` i rzutować ją, ale bez

adnotacji `@SuppressWarnings` doprowadzisz tym do ostrzeżenia w czasie kompilacji, bo taka tablica nie będzie ani przechowywać, ani dynamicznie kontrolować typu `T`. Jeśli utworzysz tablicę `String[]`, Java wymusi, aby w czasie kompilacji i w czasie wykonania mogły do niej trafiać jedynie obiekty `String`. Jednak jeśli utworzysz tablicę `Object[]`, będziesz w niej mógł umieszczać dowolne rzeczy, z wyjątkiem typów podstawowych.

Ćwiczenie 8. Zademonstruj prawdziwość twierdzeń z poprzedniego akapitu (1).

Ćwiczenie 9. Utwórz klasy niezbędne dla przykładu `Peel<Banana>` i wykaż, że kompilator ich nie przyjmie. Usuń problem przy użyciu klasy `ArrayList` (3).

Ćwiczenie 10. Zmień program *ArrayOfGenerics.java* tak, aby zamiast tablic wykorzystywał kontenery. Wykaż, że taka zmiana eliminuje ostrzeżenia kompilatora (2).

Wytwarzanie danych testowych

Przy eksperymentowaniu z tablicami (i ogólnie — z programami) przydałaby się możliwość łatwego generowania tablic wypełnionych danymi testowymi. Niniejszy podrozdział będzie więc poświęcony narzędziom ułatwiającym wypełnianie tablic wartościami bądź obiektami.

Metoda Arrays.fill()

Klasa `Arrays` ze standardowej biblioteki Javy posiada metodę `fill()`, lecz jest ona raczej prosta — powiela jedynie tę samą wartość w każdej z komórek tabeli lub w przypadku obiektów kopiuje wszędzie tę samą referencję. Oto przykład:

```
// : arrays/FillingArrays.java
// Użycie metody Arrays.fill()
import java.util.*;
import static net.mindview.util.Print.*;

public class FillingArrays {
  public static void main(String[] args) {
    int size = 6;
    boolean[] a1 = new boolean[size];
    byte[] a2 = new byte[size];
    char[] a3 = new char[size];
    short[] a4 = new short[size];
    int[] a5 = new int[size];
    long[] a6 = new long[size];
    float[] a7 = new float[size];
    double[] a8 = new double[size];
    String[] a9 = new String[size];
    Arrays.fill(a1, true);
    print("a1 = " + Arrays.toString(a1));
    Arrays.fill(a2, (byte)11);
    print("a2 = " + Arrays.toString(a2));
    Arrays.fill(a3, 'x');
    print("a3 = " + Arrays.toString(a3));
    Arrays.fill(a4, (short)17);
```

```
        print("a4 = " + Arrays.toString(a4));
        Arrays.fill(a5, 19);
        print("a5 = " + Arrays.toString(a5));
        Arrays.fill(a6, 23);
        print("a6 = " + Arrays.toString(a6));
        Arrays.fill(a7, 29);
        print("a7 = " + Arrays.toString(a7));
        Arrays.fill(a8, 47);
        print("a8 = " + Arrays.toString(a8));
        Arrays.fill(a9, "Ahoj");
        print("a9 = " + Arrays.toString(a9));
        // Manipulowanie zakresami:
        Arrays.fill(a9, 3, 5, "tam");
        print("a9 = " + Arrays.toString(a9));
    }
} /* Output:
a1 = [true, true, true, true, true, true]
a2 = [11, 11, 11, 11, 11, 11]
a3 = [x, x, x, x, x, x]
a4 = [17, 17, 17, 17, 17, 17]
a5 = [19, 19, 19, 19, 19, 19]
a6 = [23, 23, 23, 23, 23, 23]
a7 = [29.0, 29.0, 29.0, 29.0, 29.0, 29.0]
a8 = [47.0, 47.0, 47.0, 47.0, 47.0, 47.0]
a9 = [Ahoj, Ahoj, Ahoj, Ahoj, Ahoj, Ahoj]
a9 = [Ahoj, Ahoj, Ahoj, tam, tam, tam]
*///:~
```

Można albo wypełnić całą tablicę, albo — jak pokazują dwie ostatnie instrukcje — jej elementy z pewnego zakresu. Ale skoro w obu przypadkach wypełnianie sprowadza się do wstawiania tej samej wartości, użyteczność metody `Arrays.fill()` jest mocno ograniczona.

Generatory danych

Aby wypełnić tablice bardziej urozmaiconymi danymi, z zachowaniem elastyczności, skorzystamy z generatora — pojęcia prezentowanego w rozdziale „Typy ogólne". Jeśli narzędzie używa generatora `Generator`, może wygenerować dowolne dane, których charakter będzie zależny od wyboru konkretnego generatora (to przykład realizacji wzorca projektowego *Strategy* — każdy generator reprezentuje tu osobną strategię[1]).

Spróbujemy tu zdefiniować kilka generatorów; przekonałeś się już zresztą, że tworzenie własnych generatorów nie jest wielką sztuką.

Na pierwszy ogień pójdzie podstawowy zestaw generatorów zliczających dla wszystkich „opakowań" typów podstawowych i dla typu `String`. Klasy z tego zestawu będą zagnieżdżone w klasie `CountingGenerator`, tak aby typ generowanych wartości był odzwierciedlony w nazwie klasy; na przykład generator tworzący obiekty `Integer` będzie konstruowany wyrażeniem `new CountingGenerator.Integer()`:

[1] Choć nie jest do końca oczywiste. Można by bowiem uznać również, że `Generator` to realizacja wzorca projektowego *Command*. Myślę jednak, że zadanie (część zmienna we wzorcu *Command*) jest tu określone i niezmienne — chodzi o wypełnienie tablicy; różnicowany jest sposób wypełniania, więc chyba bliżej tu jednak do strategii.

```
// : net/mindview/util/CountingGenerator.java
// Proste implementacje generatorów.
package net.mindview.util;

public class CountingGenerator {
  public static class
  Boolean implements Generator<java.lang.Boolean> {
    private boolean value = false;
    public java.lang.Boolean next() {
      value = !value; // proste "przerzucanie" wartości logicznej
      return value;
    }
  }
  public static class
  Byte implements Generator<java.lang.Byte> {
    private byte value = 0;
    public java.lang.Byte next() { return value++; }
  }
  static char[] chars = ("abcdefghijklmnopqrstuvwxyz" +
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ").toCharArray();
  public static class
  Character implements Generator<java.lang.Character> {
    int index = -1;
    public java.lang.Character next() {
      index = (index + 1) % chars.length;
      return chars[index];
    }
  }
  public static class
  String implements Generator<java.lang.String> {
    private int length = 7;
    Generator<java.lang.Character> cg = new Character();
    public String() {}
    public String(int length) { this.length = length; }
    public java.lang.String next() {
      char[] buf = new char[length];
      for(int i = 0; i < length; i++)
        buf[i] = cg.next();
      return new java.lang.String(buf);
    }
  }
  public static class
  Short implements Generator<java.lang.Short> {
    private short value = 0;
    public java.lang.Short next() { return value++; }
  }
  public static class
  Integer implements Generator<java.lang.Integer> {
    private int value = 0;
    public java.lang.Integer next() { return value++; }
  }
  public static class
  Long implements Generator<java.lang.Long> {
    private long value = 0;
    public java.lang.Long next() { return value++; }
  }
  public static class
  Float implements Generator<java.lang.Float> {
```

```
    private float value = 0;
    public java.lang.Float next() {
      float result = value;
      value += 1.0;
      return result;
    }
  }
  public static class
  Double implements Generator<java.lang.Double> {
    private double value = 0.0;
    public java.lang.Double next() {
      double result = value;
      value += 1.0;
      return result;
    }
  }
} ///:~
```

Każda klasa zagnieżdżona implementuje jakiś tryb „zliczania". W przypadku klasy `CountingGenerator.Character` chodzi o wyliczanie kolejnych liter alfabetu łacińskiego — małych i wielkich, na przemian. Klasa `CountingGenerator.String` wykorzystuje generator znaków `CountingGenerator.Character` do wypełnienia tablicy znaków, a następnie konwertuje tę tablicę na ciąg `String`. Rozmiar tablicy jest podawany w wywołaniu konstruktora generatora znaków. Zauważ, że `CountingGenerator.String` w miejsce konkretnej referencji `CountingGenerator.Character` wykorzystuje typ `Generator<java.lang.Character>`. Chodzi o to, aby w przyszłości, w pliku *RandomGenerator.java*, dało się zastąpić ten generator przez `RandomGenerator.String`.

Oto narzędzie testowe, które w oparciu o refleksje operuje na idiomie zagnieżdżonej klasy generatora, dzięki czemu może testować dowolne zestawy generatorów naśladujących ten idiom:

```
// arrays/GeneratorsTest.java
import net.mindview.util.*;

public class GeneratorsTest {
  public static int size = 10;
  public static void test(Class<?> surroundingClass) {
    for(Class<?> type : surroundingClass.getClasses()) {
      System.out.print(type.getSimpleName() + ": ");
      try {
        Generator<?> g = (Generator<?>)type.newInstance();
        for(int i = 0; i < size; i++)
          System.out.printf(g.next() + " ");
        System.out.println();
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
    }
  }
  public static void main(String[] args) {
    test(CountingGenerator.class);
  }
} /* Output:
Boolean: true false true false true false true false true false
Byte: 0 1 2 3 4 5 6 7 8 9
```

```
Character: a b c d e f g h i j
String: abcdefg hijklmn opqrstu vwxyzAB CDEFGHI JKLMNOP QRSTUVW XYZabcd efghijk lmnopqr
Short: 0 1 2 3 4 5 6 7 8 9
Integer: 0 1 2 3 4 5 6 7 8 9
Long: 0 1 2 3 4 5 6 7 8 9
Float: 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
Double: 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
*///:~
```

Zakładamy tu, że testowana klasa zawiera zestaw zagnieżdżonych klas `Generator`, z których każda posiada konstruktor domyślny (czyli bezargumentowy). Metoda `getClasses()` zwraca komplet klas zagnieżdżonych. Metoda `test()` tworzy zaś egzemplarz każdej z klas i wypisuje wyniki uzyskane z wywołania metody `next()` na rzecz tych egzemplarzy.

Poniżej definiowany jest zestaw generatorów bazujących na generatorze liczb pseudolosowych. Ponieważ konstruktor klasy `Random` jest inicjalizowany wartością stałą, wyniki generowane przez poniższy program dają się powtórzyć przy każdym uruchomieniu programu:

```
//: net/mindview/util/RandomGenerator.java
// Generatory wartości losowych.
package net.mindview.util;
import java.util.*;

public class RandomGenerator {
  private static Random r = new Random(47);
  public static class
  Boolean implements Generator<java.lang.Boolean> {
    public java.lang.Boolean next() {
      return r.nextBoolean();
    }
  }
  public static class
  Byte implements Generator<java.lang.Byte> {
    public java.lang.Byte next() {
      return (byte)r.nextInt();
    }
  }
  public static class
  Character implements Generator<java.lang.Character> {
    public java.lang.Character next() {
      return CountingGenerator.chars[
        r.nextInt(CountingGenerator.chars.length)];
    }
  }
  public static class
  String extends CountingGenerator.String {
    // Podłączenie generatora losowego dla typu Character:
    { cg = new Character(); } // Inicjalizator egzemplarza
    public String() {}
    public String(int length) { super(length); }
  }
  public static class
  Short implements Generator<java.lang.Short> {
    public java.lang.Short next() {
      return (short)r.nextInt();
    }
```

```
  }
  public static class
  Integer implements Generator<java.lang.Integer> {
    private int mod = 10000;
    public Integer() {}
    public Integer(int modulo) { mod = modulo; }
    public java.lang.Integer next() {
      return r.nextInt(mod);
    }
  }
  public static class
  Long implements Generator<java.lang.Long> {
    private int mod = 10000;
    public Long() {}
    public Long(int modulo) { mod = modulo; }
    public java.lang.Long next() {
      return new java.lang.Long(r.nextInt(mod));
    }
  }
  public static class
  Float implements Generator<java.lang.Float> {
    public java.lang.Float next() {
      // Obcięcie części ułamkowej do dwóch cyfr po przecinku:
      int trimmed = Math.round(r.nextFloat() * 100);
      return ((float)trimmed) / 100;
    }
  }
  public static class
  Double implements Generator<java.lang.Double> {
    public java.lang.Double next() {
      long trimmed = Math.round(r.nextDouble() * 100);
      return ((double)trimmed) / 100;
    }
  }
} ///:~
```

Widać, że klasa `RandomGenerator.String` dziedziczy po `CountingGenerator.String` i najzwyczajniej podłącza sobie nowy obiekt generatora dla typu `Character`.

Aby nie generować nadmiernie wielkich liczb, klasa `RandomGenerator.Integer` przyjmuje domyślny moduł 10 000, ale konstruktor przeciążony pozwala narzucić mniejszy zakres. Ta sama technika wykorzystywana jest w klasie `RandomGenerator.Long`. W przypadku generatorów losowych dla typów `Float` i `Double` zajmujemy się dodatkowo przycinaniem wartości ułamkowych do pożądanej precyzji.

Do testu nowych generatorów możemy ponownie użyć klasy `GeneratorsTest`:

```
//: arrays/RandomGeneratorsTest.java
import net.mindview.util.*;

public class RandomGeneratorsTest {
  public static void main(String[] args) {
    GeneratorsTest.test(RandomGenerator.class);
  }
} /* Output:
Boolean: true false true false false true false false true true
Byte: 33 -64 -114 123 -110 -36 92 26 -96 84
```

```
Character: e G Z M m J M R o E
String: suEcUOn eOEdLsm wHLGEah KcxrEqU CBbkIna MesbtWH kjUrUkZ PgwsqPz DyCyRFJ
QAHxxHv
Short: 22219 -1560 30539 -31040 -30280 -28091 16842 -2489 15279 3284
Integer: 5382 4434 1862 8106 4332 1718 1555 1965 2760 6416
Long: 6228 8217 6051 3677 8305 4146 6862 3552 7292 7244
Float: 0.02 0.39 0.79 0.65 0.81 0.78 0.61 0.45 0.23 0.9
Double: 0.23 0.81 0.95 0.23 0.36 0.45 0.01 0.73 0.95 0.68
*///:~
```

Liczbę generowanych wartości można ustalać za pomocą pola `GeneratorsTest.size` — jest to pole publiczne.

Tworzenie tablic za pomocą generatorów

Aby za pomocą obiektu Generator utworzyć i wypełnić tablicę, potrzebujemy dwóch narzędzi konwersji. Pierwsze z nich użyje generatora do wygenerowania tablicy podtypów klasy `Object`. Drugie narzędzie będzie obsługiwać typy podstawowe — na podstawie dowolnej tablicy typów opakowujących typy podstawowe wygeneruje tablicę wartości podstawowych.

Pierwsze narzędzie będzie miało dwie opcje reprezentowane przeciążoną metodą statyczną `array()`. Pierwsza wersja tej metody przyjmie istniejący obiekt tablicy i wypełni ją przy wykorzystaniu obiektu `Generator`, zaś wersja druga przyjmie obiekt `Class`, obiekt `Generator` i pożądaną liczbę elementów oraz utworzy nową tablicę pożądanej wielkości, również wypełniając ją wartościami zwracanymi przez `Generator`. Zauważ, że narzędzie to generuje jedynie tablice podtypów `Object` i nie nadaje się do tworzenia tablic wartości podstawowych:

```
//: net/mindview/util/Generated.java
package net.mindview.util;
import java.util.*;

public class Generated {
  // Wypełnienie istniejącej tablicy:
  public static <T> T[] array(T[] a, Generator<T> gen) {
    return new CollectionData<T>(gen, a.length).toArray(a);
  }
  // Utworzenie nowej tablicy:
  @SuppressWarnings("unchecked")
  public static <T> T[] array(Class<T> type,
      Generator<T> gen, int size) {
    T[] a =
      (T[])java.lang.reflect.Array.newInstance(type, size);
    return new CollectionData<T>(gen, size).toArray(a);
  }
} ///:~
```

Klasa `CollectionData` zostanie zdefiniowana w rozdziale „Kontenery z bliska". Tworzy ona obiekt `Collection` wypełniony elementami wygenerowanymi przez generator `gen`. Liczba elementów jest określana drugim argumentem wywołania konstruktora klasy. Wszystkie podtypy `Collection` posiadają metodę `toArray()` wypełniającą przekazaną tablicę elementami przechowywanymi w obiekcie `Collection`, na rzecz którego nastąpiło wywołanie `toArray()`.

Druga wersja metody dynamicznie tworzy nowy obiekt tablicy odpowiedniego typu i rozmiaru przy użyciu mechanizmu refleksji, a potem wypełnia ją techniką identyczną jak w pierwszej wersji.

Klasę `Generator` możemy przetestować przy użyciu jednej z klas `CountingGenerator` zdefiniowanych w poprzednim punkcie:

```
//: arrays/TestGenerated.java
import java.util.*;
import net.mindview.util.*;

public class TestGenerated {
  public static void main(String[] args) {
    Integer[] a = { 9, 8, 7, 6 };
    System.out.println(Arrays.toString(a));
    a = Generated.array(a,new CountingGenerator.Integer());
    System.out.println(Arrays.toString(a));
    Integer[] b = Generated.array(Integer.class,
        new CountingGenerator.Integer(), 15);
    System.out.println(Arrays.toString(b));
  }
} /* Output:
[9, 8, 7, 6]
[0, 1, 2, 3]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
*///:~
```

Choć tablica `a` jest inicjalizowana w miejscu utworzenia, wartości elementów są potem zamazywane nowymi, a to za sprawą metody `Generated.array()`, która wstawia do tablicy nowe wartości (ale nie ingeruje w sam obiekt tablicy). Inicjalizacja tablicy `b` pokazuje zaś sposób tworzenia i wypełniania tablicy od podstaw.

Nie można stosować typów ogólnych z typami podstawowymi, a chcemy wykorzystać generatory do wypełnienia tablicy elementów takiego typu. Problem rozwiązujemy przez utworzenie konwertera, który przyjmuje tablicę obiektów opakowujących wartości podstawowe, a zwraca tablicę odpowiednich wartości podstawowych. Bez tego narzędzia musielibyśmy w generatorach przewidzieć specjalny tryb dla wszystkich typów podstawowych.

```
//: net/mindview/util/ConvertTo.java
package net.mindview.util;

public class ConvertTo {
  public static boolean[] primitive(Boolean[] in) {
    boolean[] result = new boolean[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i]; // Autounboxing
    return result;
  }
  public static char[] primitive(Character[] in) {
    char[] result = new char[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
```

```
  public static byte[] primitive(Byte[] in) {
    byte[] result = new byte[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  public static short[] primitive(Short[] in) {
    short[] result = new snort[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  public static int[] primitive(Integer[] in) {
    int[] result = new int[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  public static long[] primitive(Long[] in) {
    long[] result = new long[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  public static float[] primitive(Float[] in) {
    float[] result = new float[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  public static double[] primitive(Double[] in) {
    double[] result = new double[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
} ///:~
```

Każda z wersji metody `primitive()` tworzy tablicę wartości podstawowych odpowiedniego typu i rozmiaru, a następnie kopiuje elementy z tablicy `in` przechowującej obiekty opakowujące wartości podstawowe. Zwróć uwagę na wykorzystanie mechanizmu pakowania w obiekty w wyrażeniu:

```
result[i] = in[i];
```

Oto przykład, który pokazuje możliwość użycia klasy `ConvertTo` z obiema wersjami metody `Generated.array()`:

```
//: arrays/PrimitiveConversionDemonstration.java
import java.util.*;
import net.mindview.util.*;

public class PrimitiveConversionDemonstration {
  public static void main(String[] args) {
    Integer[] a = Generated.array(Integer.class,
        new CountingGenerator.Integer(), 15);
    int[] b = ConvertTo.primitive(a);
    System.out.println(Arrays.toString(b));
```

```
    boolean[] c = ConvertTo.primitive(
      Generated.array(Boolean.class,
        new CountingGenerator.Boolean(), 7));
    System.out.println(Arrays.toString(c));
  }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
[true, false, true, false, true, false, true]
*///:~
```

Dotarliśmy wreszcie do programu, który testuje technikę tworzenia tablic przy użyciu klas generatora `RandomGenerator`:

```
//: arrays/TestArrayGeneration.java
// Test narzędzi wypełniających tablicę przy użyciu generatorów.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class TestArrayGeneration {
  public static void main(String[] args) {
    int size = 6;
    boolean[] a1 = ConvertTo.primitive(Generated.array(
      Boolean.class, new RandomGenerator.Boolean(), size));
    print("a1 = " + Arrays.toString(a1));
    byte[] a2 = ConvertTo.primitive(Generated.array(
      Byte.class, new RandomGenerator.Byte(), size));
    print("a2 = " + Arrays.toString(a2));
    char[] a3 = ConvertTo.primitive(Generated.array(
      Character.class,
      new RandomGenerator.Character(), size));
    print("a3 = " + Arrays.toString(a3));
    short[] a4 = ConvertTo.primitive(Generated.array(
      Short.class, new RandomGenerator.Short(), size));
    print("a4 = " + Arrays.toString(a4));
    int[] a5 = ConvertTo.primitive(Generated.array(
      Integer.class, new RandomGenerator.Integer(), size));
    print("a5 = " + Arrays.toString(a5));
    long[] a6 = ConvertTo.primitive(Generated.array(
      Long.class, new RandomGenerator.Long(), size));
    print("a6 = " + Arrays.toString(a6));
    float[] a7 = ConvertTo.primitive(Generated.array(
      Float.class, new RandomGenerator.Float(), size));
    print("a7 = " + Arrays.toString(a7));
    double[] a8 = ConvertTo.primitive(Generated.array(
      Double.class, new RandomGenerator.Double(), size));
    print("a8 = " + Arrays.toString(a8));
  }
} /* Output:
a1 = [true, false, true, false, false, true]
a2 = [104, -79, -76, 126, 33, -64]
a3 = [Z, n, T, c, Q, r]
a4 = [-13408, 22612, 15401, 15161, -28466, -12603]
a5 = [7704, 7383, 7706, 575, 8410, 6342]
a6 = [7674, 8804, 8950, 7826, 4322, 896]
a7 = [0.01, 0.2, 0.4, 0.79, 0.27, 0.45]
a8 = [0.16, 0.87, 0.7, 0.66, 0.87, 0.59]
*///:~
```

Tym samym upewniliśmy się co do poprawnego działania metod `ConvertTo.primitive()`.

Ćwiczenie 11. Wykaż, że autoboxing nie działa z tablicami (2).

Ćwiczenie 12. Utwórz za pomocą generatora `CountingGenerator` zainicjalizowaną tablicę elementów typu `double` (1).

Ćwiczenie 13. Wypełnij ciąg `String` za pomocą generatora `CountingGenerator.Character` (2).

Ćwiczenie 14. Utwórz tablicę dla każdego z typów podstawowych, a potem wypełnij każdą z nich przy użyciu generatora `CountingGenerator`. Wypisz zawartość tablic (6).

Ćwiczenie 15. Zmodyfikuj program `ContainerComparison.java` przez utworzenie w nim generatora dla `BerylliumSphere` i zmień metodę `main()` tak, aby wykorzystywała ten generator i metodę `Generated.array()` (2).

Ćwiczenie 16. Bazując na programie *CountingGenerator.java*, napisz klasę `Skip-Generator`, która będzie generować wartości szeregu arytmetycznego o różnicy definiowanej argumentem konstruktora. Zmodyfikuj program *TestArrayGeneration.java* tak, aby pokazać prawidłowe działanie nowej klasy (3).

Ćwiczenie 17. Utwórz i przetestuj klasę generatora dla typu `BigDecimal` i upewnij się, że działa z metodami klasy `Generated` (5).

Narzędzia klasy Arrays

W pakiecie *java.util* można znaleźć klasę o nazwie `Arrays`, posiadającą zestaw metod statycznych, które są przydatne do wykonywania pewnych operacji na tablicach. Mamy tam cztery podstawowe funkcje: `equals()` do sprawdzania równości dwóch tablic (i `deepEquals()` dla tablic wielowymiarowych), `fill()` do wypełniania tablicy określoną wartością, `sort()` do sortowania tablicy oraz `binarySearch()` do wyszukiwania elementu w tablicy posortowanej; do tego `toString()` do wytwarzania ciągu znaków zawierającego reprezentację tablicy i `hashCode()` do obliczania wartości skrótu (ang. *hash code*) dla tablicy (o skrótach i haszowaniu dowiesz się więcej w rozdziale „Kontenery z bliska"). Każda z tych metod ma wersje przeciążone dla każdego z typów podstawowych i klasy `Object`. Poza tym jest jeszcze metoda `asList()`, która zamienia dowolną tablicę w kontener `List` — poznałeś go w rozdziale „Kolekcje obiektów".

Zanim przejdziemy do omawiania metod klasy `Arrays()`, przyjrzymy się pewnej przydatnej metodzie, która nie wchodzi w skład tej klasy.

Kopiowanie tablic

Standardowa biblioteka Javy dostarcza statyczną metodę `System.arraycopy()`, która pozwala uzyskać znacznie szybsze kopiowanie tablicy, niż gdybyśmy przeprowadzili to, używając własnej pętli `for`. Metoda `System.arraycopy()` jest przeciążona tak, że obsługuje tablice wszystkich typów. Oto przykład korzystający z tablic typu `int`:

```
// : arrays/CopyingArrays.java
// Użycie metody System.arraycopy()
import java.util.*;
import static net.mindview.util.Print.*;

public class CopyingArrays {
  public static void main(String[] args) {
    int[] i = new int[7];
    int[] j = new int[10];
    Arrays.fill(i, 47);
    Arrays.fill(j, 99);
    print("i = " + Arrays.toString(i));
    print("j = " + Arrays.toString(j));
    System.arraycopy(i, 0, j, 0, i.length);
    print("j = " + Arrays.toString(j));
    int[] k = new int[5];
    Arrays.fill(k, 103);
    System.arraycopy(i, 0, k, 0, k.length);
    print("k = " + Arrays.toString(k));
    Arrays.fill(k, 103);
    System.arraycopy(k, 0, i, 0, k.length);
    print("i = " + Arrays.toString(i));
    // Obiekty:
    Integer[] u = new Integer[10];
    Integer[] v = new Integer[5];
    Arrays.fill(u, new Integer(47));
    Arrays.fill(v, new Integer(99));
    print("u = " + Arrays.toString(u));
    print("v = " + Arrays.toString(v));
    System.arraycopy(v, 0, u, u.length/2, v.length);
    print("u = " + Arrays.toString(u));
  }
} /* Output:
i = [47, 47, 47, 47, 47, 47, 47]
j = [99, 99, 99, 99, 99, 99, 99, 99, 99, 99]
j = [47, 47, 47, 47, 47, 47, 47, 99, 99, 99]
k = [47, 47, 47, 47, 47]
i = [103, 103, 103, 103, 103, 47, 47]
u = [47, 47, 47, 47, 47, 47, 47, 47, 47, 47]
v = [99, 99, 99, 99, 99]
u = [47, 47, 47, 47, 47, 99, 99, 99, 99, 99]
*///:~
```

Parametry metody `arraycopy()` to: tablica źródłowa, przesunięcie początkowe w tablicy źródłowej, od którego należy rozpocząć kopiowanie, tablica docelowa, przesunięcie początkowe w tablicy docelowej oraz liczba elementów do przekopiowania. Naturalnie każde naruszenie zakresu tablicy spowoduje zgłoszenie wyjątku.

Przykład pokazuje, że można kopiować zarówno tablice zawierające typy podstawowe, jak i obiekty. Jednak podczas kopiowania tablic obiektów kopiowane są wyłącznie referencje — nie są tworzone kopie obiektów będących elementami tablicy. Nazywa się to *kopiowaniem płytkim* (zobacz suplementy publikowane on-line).

Metoda `System.arraycopy()` nie wykorzystuje mechanizmu automatycznego pakowania wartości podstawowych do obiektów ani ich rozpakowywania z takich obiektów — tablice (źródłowa i docelowa) muszą przechowywać elementy tego samego typu.

Ćwiczenie 18. Utwórz i wypełnij tablicę obiektów `BerylliumSphere`. Skopiuj ją do nowej tablicy i pokaż, że kopiowanie było płytkie (3).

Porównywanie tablic

Klasa `Arrays` udostępnia metodę `equals()`, która pozwala porównać całe tablice pod względem równości. Metoda ta została przeciążona dla każdego typu podstawowego oraz `Object`. Aby zachodziła równość, obie tablice muszą posiadać tę samą liczbę elementów i każdy z nich musi być równy odpowiadającym mu elementom w drugiej tablicy, zgodnie z wynikiem metody `equals()` wywołanej dla elementów (w przypadku typów podstawowych używana jest metoda `equals()` klasy opakowującej typ, na przykład metoda `Integer.equals()` dla typu `int`). Oto przykład:

```
//: arrays/ComparingArrays.java
// Użycie metody Arrays.equals()
import java.util.*;
import static net.mindview.util.Print.*;

public class ComparingArrays {
  public static void main(String[] args) {
    int[] a1 = new int[10];
    int[] a2 = new int[10];
    Arrays.fill(a1, 47);
    Arrays.fill(a2, 47);
    print(Arrays.equals(a1, a2));
    a2[3] = 11;
    print(Arrays.equals(a1, a2));
    String[] s1 = new String[4];
    Arrays.fill(s1, "Hej");
    String[] s2 = { new String("Hej"), new String("Hej"),
      new String("Hej"), new String("Hej") };
    print(Arrays.equals(s1, s2));
  }
} /* Output:
true
false
true
*///:~
```

Początkowo `a1` i `a2` są dokładnie takie same, toteż na wyjściu mamy „true", ale później jeden z elementów zostaje zmieniony, stąd w drugim wierszu wyniku dostaniemy „false". W ostatnim przypadku wszystkie elementy `s1` wskazują ten sam obiekt, natomiast `s2` zawiera pięć obiektów unikatowych. Mimo to równość tablic opiera się na zawartości (poprzez `Object.equals()`) i dlatego w wyniku dostaniemy „true".

Ćwiczenie 19. Utwórz klasę z polem `int` inicjalizowanym na podstawie argumentu konstruktora. Zbuduj dwie tablice obiektów tej klasy, wykorzystując do ich inicjalizacji identyczne wartości, i pokaż, że metoda `Arrays.equals()` stwierdza nierówność odpowiednich elementów obu tablic. Dodaj do swojej klasy metodę `equals()` tak, aby usunąć problem (2).

Ćwiczenie 20. Zademonstruj działanie `deepEquals()` dla tablic wielowymiarowych (4).

Porównywanie elementów tablic

Problem napisania ogólnego programu sortującego polega na tym, że sortowanie musi wykorzystywać porównania, bazując na faktycznym typie obiektu. Oczywiście jednym z rozwiązań jest napisanie odrębnej metody sortującej dla każdego z typów, ale jak łatwo zauważyć, nie daje to kodu możliwego do wykorzystania z nowymi typami.

Podstawową regułą projektowania oprogramowania jest „oddzielenie elementów zmiennych od tych niepodlegających zmianom". Tutaj kodem niezmiennym jest ogólny algorytm sortowania, natomiast sprawą, która się zmienia, jest sposób porównania elementów. Tak więc zamiast zaszywać na stałe kod porównujący w wielu różnych metodach sortujących, używa się wzorca projektowego *Strategy*[2]. Dzięki niemu zmienna część kodu jest hermetyzowana w osobnej klasie (obiekcie strategii). Obiekt strategii przekazuje się do kodu niezmiennego, który wykorzysta obiekt strategii do doprecyzowania własnego algorytmu. W obiektach strategii można więc wyrażać różne sposoby porównywania przy sortowaniu, współpracujące z niezmiennym ogólnym algorytmem sortowania.

W Javie funkcję porównawczą można tworzyć na dwa sposoby. Pierwszy wykorzystuje *naturalną metodę porównującą*, zamieszczoną w klasie poprzez implementację interfejsu `java.lang.Comparable`. Jest to bardzo prosty interfejs z jedyną metodą o nazwie `compareTo()`. Metoda pobiera inny obiekt tego samego typu jako argument i zwraca wartość ujemną, kiedy obiekt aktualny jest mniejszy niż argument, zero — kiedy oba obiekty są równe, i wartość dodatnią — kiedy obiekt aktualny jest większy od argumentu.

Oto klasa, która implementuje interfejs `Comparable` i pokazuje zdolność porównywania poprzez użycie metody `Arrays.sort()` ze standardowej biblioteki Javy:

```
//: arrays/CompType.java
// Implementowanie intrefejsu Comparable w klasie.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class CompType implements Comparable<CompType> {
  int i;
  int j;
  private static int count = 1;
  public CompType(int n1, int n2) {
    i = n1;
    j = n2;
  }
  public String toString() {
    String result = "[i = " + i + ", j = " + j + "]";
    if(count++ % 3 == 0)
      result += "\n";
    return result;
  }
  public int compareTo(CompType rv) {
    return (i < rv.i ? -1 : (i == rv.i ? 0 : 1));
  }
```

[2] Za *Design Patterns* autorstwa Ericha Gamma i spółki (Addison-Wesley, 1995). Zobacz też *Thinking in Patterns (with Java)* na stronach *www.MindView.net*.

```
  private static Random r = new Random(47);
  public static Generator<CompType> generator() {
    return new Generator<CompType>() {
      public CompType next() {
        return new CompType(r.nextInt(100),r.nextInt(100));
      }
    };
  }
  public static void main(String[] args) {
    CompType[] a =
      Generated.array(new CompType[12], generator());
    print("przed sortowaniem:");
    print(Arrays.toString(a));
    Arrays.sort(a);
    print("przed sortowaniu:");
    print(Arrays.toString(a));
  }
} /* Output:
przed sortowaniem:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
]
po sortowaniu:
[[i = 9, j = 78], [i = 11, j = 22], [i = 16, j = 40]
, [i = 20, j = 58], [i = 22, j = 7], [i = 51, j = 89]
, [i = 58, j = 55], [i = 61, j = 29], [i = 68, j = 0]
, [i = 88, j = 28], [i = 93, j = 61], [i = 98, j = 61]
]
*///:~
```

Kiedy definiujemy funkcję porównującą, do nas należy decyzja, co to znaczy porównać jeden obiekt z innym. Powyżej, w porównaniu, została użyta tylko zmienna `i`, a `j` pominięta.

Metoda `generator()` zwraca obiekt implementujący interfejs `Generator` poprzez stworzenie anonimowej klasy wewnętrznej (więcej w rozdziale 10.). W ten sposób tworzony jest obiekt `CompType` inicjowany wartością losową. W `main()` generator został użyty do wypełnienia tablicy obiektów `CompType`, którą następnie sortujemy. Jeśli nie zaimplementowalibyśmy interfejsu `Comparable`, to wywołanie metody `sort()` doprowadziłoby podczas wykonania programu do wygenerowania wyjątku `ClassCastException`. Byłoby to spowodowane tym, że `sort()` rzutuje argumenty wywołania do typu `Comparable`.

Przypuśćmy dalej, że ktoś podarował nam klasę, która nie implementuje `Comparable`, albo też podarował klasę *implementującą* `Comparable`, ale zdecydowaliśmy, że jej działanie nam nie odpowiada, i chcemy mieć inną funkcję porównującą dla tego typu. Rozwiązanie tego problemu nie polega na umieszczeniu kodu porównania na stałe we wszystkich typach obiektów. Rozwiązanie problemu sprowadza się do utworzenia osobnej klasy implementującej interfejs `Comparator` (wprowadzonej już w rozdziale „Kolekcje obiektów"). To kolejny przykład realizacji wzorca projektowego *Strategy*. Interfejs ten ma dwie metody: `compare()` i `equals()`. Jednak nie trzeba implementować metody `equals()` poza przypadkami chęci zwiększenia wydajności działania, gdyż za każdym razem, gdy tworzymy klasę, jest ona pośrednio wywiedziona z klasy `Object`, która zawiera już `equals()`. Można zatem po prostu użyć tej domyślnej metody `equals()` klasy `Object` i z powodzeniem wykorzystywać interfejs.

Klasa `Collections` (zajmiemy się nią w następnym rozdziale) zawiera metodę `reverseOrder()` generującą pojedynczy obiekt `Comparator`, który odwraca naturalną kolejność sortowania. Można to łatwo zastosować do klasy `CompType`:

```
//: arrays/Reverse.java
// Komparator Collections.reverseOrder()
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Reverse {
  public static void main(String[] args) {
    CompType[] a = Generated.array(
      new CompType[12], CompType.generator());
    print("przed sortowaniem:");
    print(Arrays.toString(a));
    Arrays.sort(a, Collections.reverseOrder());
    print("po sortowaniu:");
    print(Arrays.toString(a));
  }
} /* Output:
przed sortowaniem:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
]
po sortowaniu:
[[i = 98, j = 61], [i = 93, j = 61], [i = 88, j = 28]
, [i = 68, j = 0], [i = 61, j = 29], [i = 58, j = 55]
, [i = 51, j = 89], [i = 22, j = 7], [i = 20, j = 58]
, [i = 16, j = 40], [i = 11, j = 22], [i = 9, j = 78]
]
*///:~
```

Możesz też napisać własny komparator. Ten poniżej porównuje obiekty `CompType` na podstawie ich wartości `j` zamiast poprzednio wykorzystywanej `i`:

```
//: arrays/ComparatorTest.java
// Implementowanie własnego komparatora dla klasy.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class CompTypeComparator implements Comparator<CompType> {
  public int compare(CompType o1, CompType o2) {
    return (o1.j < o2.j ? -1 : (o1.j == o2.j ? 0 : 1));
  }
}

public class ComparatorTest {
  public static void main(String[] args) {
    CompType[] a = Generated.array(
      new CompType[12], CompType.generator());
    print("przed sortowaniem:");
    print(Arrays.toString(a));
    Arrays.sort(a, new CompTypeComparator());
    print("po sortowaniu:");
    print(Arrays.toString(a));
```

```
  }
} /* Output:
przed sortowaniem:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
]
po sortowaniu:
[[i = 68, j = 0], [i = 22, j = 7], [i = 11, j = 22]
, [i = 88, j = 28], [i = 61, j = 29], [i = 16, j = 40]
, [i = 58, j = 55], [i = 20, j = 58], [i = 93, j = 61]
, [i = 98, j = 61], [i = 9, j = 78], [i = 51, j = 89]
]
*///:~
```

Ćwiczenie 21. Spróbuj posortować tablicę obiektów z ćwiczenia 18. Zaimplementuj własny komparator `Comparable` w celu wyeliminowania problemu. Utwórz obiekt `Comparator` do sortowania obiektów w odwrotnej kolejności (3).

Sortowanie tablic

Dzięki wbudowanym metodom sortującym można posortować zawartość tablicy dowolnego typu podstawowego i dowolnej tablicy obiektów, które bądź implementują `Comparable`, bądź też posiadają odpowiedni `Comparator`[3]. Poniżej zamieszczam przykład, który generuje losowe ciągi tekstowe, a następnie je sortuje:

```
//: arrays/StringSorting.java
// Sortowanie tablicy ciągów String.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class StringSorting {
  public static void main(String[] args) {
    String[] sa = Generated.array(new String[20],
      new RandomGenerator.String(5));
    print("Przed sortowaniem: " + Arrays.toString(sa));
    Arrays.sort(sa);
    print("Po  sortowaniu: " + Arrays.toString(sa));
    Arrays.sort(sa, Collections.reverseOrder());
    print("Sortowanie odwrotne: " + Arrays.toString(sa));
    Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
    print("Sortowanie bez uwzględniania wielkości liter: " + Arrays.toString(sa));
  }
} /* Output:
Przed sortowaniem: [YNzbr, nyGcF, OWZnT, cQrGs, eGZMm, JMRoE, suEcU, OneOE, dLsmw, HLGEa,
hKcxr, EqUCB, bkIna, Mesbt, WHkjU, rUkZP, gwsqP, zDyCy, RFJQA, HxxHv]
Po sortowaniu: [EqUCB, HLGEa, HxxHv, JMRoE, Mesbt, OWZnT, OneOE, RFJQA, WHkjU, YNzbr,
bkIna, cQrGs, dLsmw, eGZMm, gwsqP, hKcxr, nyGcF, rUkZP, suEcU, zDyCy]
Sortowanie odwrotne: [zDyCy, suEcU, rUkZP, nyGcF, hKcxr, gwsqP, eGZMm, dLsmw, cQrGs, bkIna,
YNzbr, WHkjU, RFJQA, OneOE, OWZnT, Mesbt, JMRoE, HxxHv, HLGEa, EqUCB]
Sortowanie bez uwzględniania wielkości liter: [bkIna, cQrGs, dLsmw, eGZMm, EqUCB, gwsqP, hKcxr,
HLGEa, HxxHv, JMRoE, Mesbt, nyGcF, OneOE, OWZnT, RFJQA, rUkZP, suEcU, WHkjU, YNzbr, zDyCy]
*///:~
```

[3] Co ciekawe, w Javie 1.0 i 1.1 nie było nawet obsługi sortowania obiektów `String`.

Algorytm sortowania ciągów `String` przyjmuje domyślnie porządek *leksykograficzny*, dlatego wypisuje słowa począwszy od tych rozpoczynających się wielką literą, a dopiero po nich rozpoczynające się małymi literami (tak sortuje się zwykle zawartość książek telefonicznych). Możemy jednak również chcieć grupować słowa bez względu na wielkość liter — wystarczy użyć komparatora `String.CASE_INSENSITIVE_ORDER`, jak w powyższym przykładzie w ostatnim wywołaniu metody `sort()`.

Algorytm sortujący stosowany w standardowej bibliotece Javy został zaprojektowany tak, aby optymalnie sortował określone typy danych — sortowanie szybkie (ang. *quicksort*) dla typów *podstawowych* i stabilne sortowanie przez scalanie (ang. *merge sort*) w przypadku obiektów. Nie trzeba więc martwić się o wydajność, dopóki stosowane przez nas narzędzie sprawdzające wydajność (tzw. profiler) nie wykaże, że sortowanie stanowi wąskie gardło.

Przeszukiwanie tablicy posortowanej

Jeżeli mamy już posortowaną tablicę, można szybko wyszukać konkretny element, stosując metodę `Arrays.binarySearch()`. Ale jeśli użyjesz tej metody do przeszukania tablicy nieposortowanej, wyniki będą niewiarygodne i nieprzewidywalne. Poniższy przykład wykorzystuje klasę `RandomGenerator.Integer` do wypełnienia tablicy oraz wylosowania wartości, a potem ten sam generator wykorzystuje ponownie, do wybrania wartości wyszukiwanych:

```
// : arrays/ArraySearching.java
// Użycie metody Arrays.binarySearch().
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ArraySearching {
  public static void main(String[] args) {
    Generator<Integer> gen =
      new RandomGenerator.Integer(1000);
    int[] a = ConvertTo.primitive(
      Generated.array(new Integer[25], gen));
    Arrays.sort(a);
    print("Tablica posortowana: " + Arrays.toString(a));
    while(true) {
      int r = gen.next();
      int location = Arrays.binarySearch(a, r);
      if(location >= 0) {
        print("Pozycja wartości " + r + " to " + location +
          ", a[" + location + "] = " + a[location]);
        break; // Wyjście poza pętlę
      }
    }
  }
} /* Output:
Tablica posortowana: [128, 140, 200, 207, 258, 258, 278, 288, 322, 429, 511, 520, 522, 551, 555, 589, 693,
704, 809, 861, 861, 868, 916, 961, 998]
Pozycja wartości 322 to 8, a[8] = 322
*///:~
```

W pętli `while` generowane są, jako wartości losowe, elementy do wyszukania, dopóki jakiś nie zostanie odnaleziony.

Jeżeli szukany element zostanie odnaleziony, metoda `Arrays.binarySearch()` zwróci wartość większą lub równą zero. W przeciwnym razie dostaniemy wartość ujemną, oznaczającą miejsce, w którym element powinien być wstawiony, gdybyśmy chcieli to zrobić. Wartość ta to dokładnie:

```
-(punkt wstawienia) - 1
```

Punkt wstawienia jest indeksem pierwszego elementu większego niż szukany lub wartością `a.size()`, gdy wszystkie elementy tablicy są mniejsze niż poszukiwany.

W przypadku tablicy zawierającej duplikaty wartości nie mamy żadnej pewności, który z takich duplikatów zostanie odnaleziony. Zatem algorytm nie jest naprawdę opracowany dla przypadków tablic z elementami powtarzającymi się, choć je toleruje. Jeżeli potrzebna nam posortowana lista niepowtarzalnych elementów, to można użyć kontenera `TreeSet` (aby elementy były posortowane) lub `LinkedHashSet` (aby zachować kolejność, w jakiej elementy były dodawane). Te klasy automatycznie zadbają o wszystkie szczegóły za nas. Tylko w przypadku wyraźnego spadku wydajności powinno się zastąpić je własnoręcznie zarządzaną tablicą.

Jeśli posortujemy tablicę obiektów, stosując `Comparator` (tablice typów podstawowych nie pozwalają na takie sortowanie), to trzeba włączyć ten `Comparator` do szukania metodą `binarySearch()` (stosując przeciążoną wersję `binarySearch()`). Zmodyfikujmy dla przykładu program *StringSorting.java* tak, by można przeprowadzić wyszukiwanie:

```
//: arrays/AlphabeticSearch.java
// Wyszukiwanie przy użyciu komparatora.
import java.util.*;
import net.mindview.util.*;

public class AlphabeticSearch {
  public static void main(String[] args) {
    String[] sa = Generated.array(new String[30],
      new RandomGenerator.String(5));
    Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
    System.out.println(Arrays.toString(sa));
    int index = Arrays.binarySearch(sa, sa[10],
      String.CASE_INSENSITIVE_ORDER);
    System.out.println("Indeks: "+ index + "\n"+ sa[index]);
  }
} /* Output:
[bkIna, cQrGs, cXZJo, dLsmw, eGZMm, EqUCB, gwsqP, hKcxr, HLGEa, HqXum, HxxHv, JMRoE, JmzMs,
Mesbt, MNvqe, nyGcF, ogoYW, OneOE, OWZnT, RFJQA, rUkZP, sgqia, slJrL, suEcU, uTpnX, vpfFv,
WHkjU, xxEAJ, YNzbr, zDyCy]
Indeks: 10
HxxHv
*///:~
```

`Comparator` musi być przekazany do przeciążonej metody `binarySearch()` jako jej trzeci argument. Powyższy przykład zawsze znajdzie element, ponieważ szukany element jest pobierany z tej samej tablicy, którą przeszukujemy.

Ćwiczenie 22. Wykaż, że wynik użycia metody `binarySearch()` na tablicy nieposortowanej jest nieprzewidywalny (2).

Ćwiczenie 23. Utwórz tablicę obiektów `Integer`, wypełnij ją losowymi wartościami typu `int` (korzystając z mechanizmu automatycznego pakowania wartości podstawowych w obiekty) i posortuj zawartość tablicy w kolejności odwrotnej za pomocą komparatora `Comparator` (2).

Ćwiczenie 24. Wykaż, że klasa z ćwiczenia 19. może skutecznie podlegać operacji przeszukiwania (3).

Podsumowanie

Lektura tego rozdziału przekonała Cię zapewne, że Java dysponuje sensowną obsługą niskopoziomowej struktury danych, jaką jest tablica o ustalonym rozmiarze. Tego rodzaju tablica ma przedkładać wydajność nad elastyczność, dokładnie jak w modelu tablic w językach C i C++. W początkowej wersji języka Java niskopoziomowe tablice o stałym rozmiarze były absolutnie niezbędne nie tylko dlatego, że architekci języka postanowili włączyć do niego typy podstawowe (również ze względu na wydajność), ale i dlatego, że ówczesna obsługa kontenerów była szczątkowa. W pierwszych wydaniach Javy stosowanie tablic było więc normą.

W kolejnych wydaniach języka polepszono dostępność i obsługę kontenerów, które teraz zaczynają wyprzedzać tablice na wszystkich polach, z wyjątkiem może wydajności, choć i pod tym względem kontenery są coraz sprawniejsze. Wielokrotnie zresztą w tej książce podkreślam, że problemy z wydajnością i tak pojawiają się niemal zawsze tam, gdzie się ich zupełnie nie spodziewamy.

Dzięki mechanizmowi pakowania wartości podstawowych w obiekty odpowiedniego typu można już wygodnie umieszczać w kontenerach wartości typów podstawowych, co stanowi kolejny czynnik wypierający tablice. A ponieważ typy ogólne pozwalają na stosowanie bezpiecznych, typowanych kontenerów, tablice doczekały się godnego konkurenta również i na tym polu.

W rozdziale pokazywałem, że typy ogólne nie lubią się specjalnie z tablicami. Często jeśli już można zmusić je do jakiejś formy współpracy (zobacz następny rozdział), kończy się przynajmniej na ostrzeżeniach „unchecked" kompilatora.

Kilkukrotnie miałem okazję słyszeć od projektantów Javy, że powinienem korzystać z kontenerów, a nie tablic; porada taka wypływała zwykle przy dyskutowaniu konkretnych przykładów (używałem tablic do demonstrowania specyficznych technik i nie miałem innej opcji).

Wszystko to każe sądzić, że faktycznie powinniśmy jako programiści preferować kontenery względem tablic. I tylko tam, gdzie wydajność operacji jest absolutnym priorytetem (a zastosowanie tablicy w miejsce kontenera nie stanowi większego problemu projektowego), możemy uciekać się do stosowania tablic.

Tymczasem niektóre języki w ogóle nie posiadają odpowiedników prostych tablic o stałym rozmiarze. Dysponują jedynie kontenerami o dynamicznie dostosowywanym rozmiarze i to znacznie rozbudowanych wobec tablic znanych z języków C, C++ czy Java. Na przykład w języku Python[4] istnieje typ list obsługiwany za pomocą składni typowej dla tablic, ale o znacznie większym zestawie funkcji — można nawet po nim dziedziczyć:

```
#: arrays/PythonLists.py

aList = [1, 2, 3, 4, 5]
print type(aList) # <type 'list'>
print aList # [1, 2, 3, 4, 5]
print aList[4] # 5  Proste indeksowanie list
aList.append(6) # Listy można wydłużać
print aList += [7, 8] # [1, 2, 3, 4, 5, 6, 7, 8]
aSlice = aList[2:4]
print aSlice # [3, 4]

class MyList(list): # Dziedziczenie po typie list
   # Definiowanie metody, wskaźnik 'this' jest tu jawny
   def getReversed(self):
      reversed = self[:] # Kopiowanie listy przekrojami
      reversed.reverse() # Wbudowana metoda typu list
      return reversed

list2 = MyList(aList) # Przy tworzeniu obiektu niepotrzebne 'new'
print type(list2) # <class '__main__.MyList'>
print list2.getReversed() # [8, 7, 6, 5, 4, 3, 2, 1]
#:~
```

Podstawowa składnia języka Python została zaprezentowana w poprzednim rozdziale. W powyższym przykładzie mamy do czynienia z utworzeniem listy przez proste podanie sekwencji obiektów ujętych w nawiasy prostokątne. Wynikowy obiekt ma typ czasu wykonania `list` (ujawnia to instrukcja `print`; wyniki tej instrukcji są podawane w komentarzach). Efekt wypisania samej listy jest podobny jak efekt wywołania metody `Arrays.toString()` w języku Java.

Tworzenie podsekwencji listy polega na „wykrawaniu", realizowanym za pośrednictwem operatora : umieszczanego w nawiasie indeksującym. Typ list ma zresztą znacznie więcej własnych operatorów.

`MyList` to definicja klasy; klasa ta dziedziczy po klasie wymienionej w nawiasie. Wewnątrz klasy instrukcja `def` definiuje metody; pierwszym argumentem metody jest zawsze obiekt, na rzecz którego wywołuje się metodę — w Javie jest to `this` i jest przekazywany niejawnie. W języku Python jest on jawny i występuje zwykle jako identyfikator `self` (choć nie jest to słowo kluczowe języka). Zauważ, że klasa automatycznie dziedziczy konstruktor.

Choć wszystko w Pythonie jest obiektem (dotyczy to nawet typów liczbowych — całkowitych i zmiennoprzecinkowych), programiści stojący przed zadaniem polepszenia efektywności operacji mogą uciekać się do rozszerzeń pisanych w językach C, C++ albo tworzonych

[4] Zobacz *www.Python.org*.

za pomocą specjalnego narzędzia o nazwie Pyrex, służącego właśnie do przyspieszania kodu. W ten sposób godzi się czystość programowania obiektowego z konieczną niekiedy wydajnością.

Język PHP 5[5] idzie jeszcze dalej, udostępniając zaledwie jeden typ tablicowy dający się indeksować wartościami liczbowymi, ale pełniący równocześnie rolę tablicy asocjacyjnej (jak kontener `Map` w Javie).

Po tylu latach rozwoju języka Java można spekulować, czy gdyby jego projektanci zaczynali dziś pracę od zera, to umieściliby w nim niskopoziomowe elementy, jak tablice i typy podstawowe. Gdyby ich zabrakło, język byłby już czysto obiektowy (bo w obecnej postaci Java, mimo wszystko, nie jest językiem czysto obiektowym, choćby właśnie ze względu na elementy niskopoziomowe). Argument o wydajności jest co prawda przekonujący, ale i w tej dziedzinie obserwuje się zmianę trendu — coraz częściej nad wydajność przedkłada się elastyczność oferowaną przez wysokopoziomowe komponenty, jak kontenery. A gdyby do tego kontenery zostałyby wbudowane do rdzenia języka (jak w niektórych innych językach), kompilator miałby większe możliwości optymalizacji.

Odłóżmy spekulacje na bok, bo rzeczywistość wciąż jest domeną tablic i wciąż trudno się bez nich obejść. Ale generalnie kontenery to coraz lepsza alternatywa.

Ćwiczenie 25. Przepisz program *PythonLists.py* w języku Java (3).

Rozwiązania wybranych ćwiczeń można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide* dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

[5] Zobacz *www.php.net*.

Rozdział 17.

Kontenery z bliska

Rozdział „Kolekcje obiektów" przedstawił podstawowe koncepcje i pokazał najważniejsze funkcje biblioteki kontenerów Javy w stopniu wystarczającym do korzystania z kontenerów we własnych aplikacjach. Tym razem przyjrzymy się bibliotece kontenerów z bliska.

Aby w pełni wykorzystać możliwości biblioteki kontenerów, trzeba rozszerzyć wiedzę o kontenerach poza to, co zostało podane w rozdziale „Kolekcje obiektów". Opóźnienie przekazania tej dawki wiedzy do tego momentu było konieczne choćby ze względu na fakt, że wcześniej należało rozprawić się z typami ogólnymi.

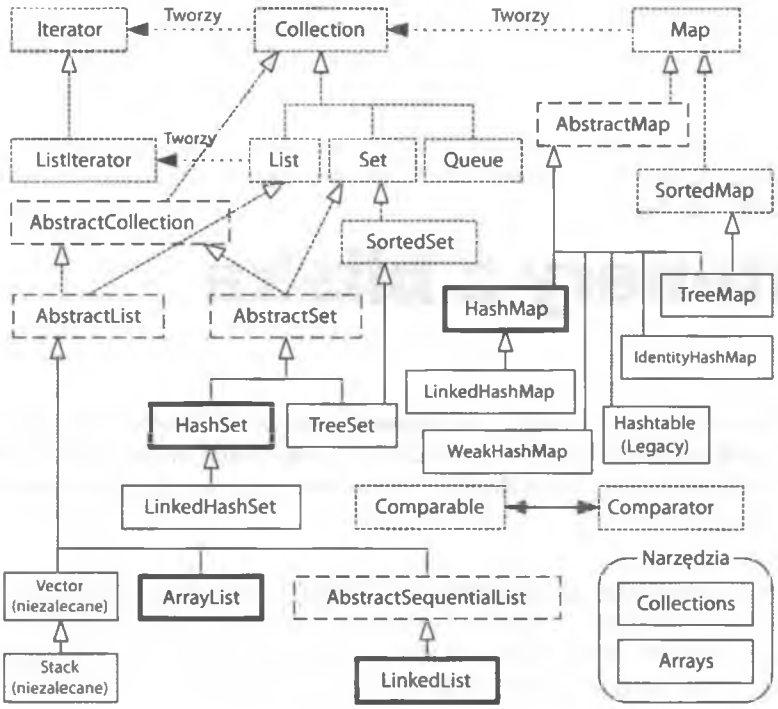
Niniejszy, rozszerzony przegląd cech i możliwości kontenerów obejmuje między innymi haszowanie, w tym sposób pisania własnych metod `hashCode()` i `equals()` pod kątem kontenerów haszowanych. Dowiesz się tutaj też o różnych odmianach poszczególnych kontenerów i nauczysz się wybierać je odpowiednio do kontekstu zastosowania. Rozdział zakończy się przeglądem narzędzi ogólnego przeznaczenia i klas specjalnych.

Pełna taksonomia kontenerów

W podsumowaniu rozdziału „Kolekcje obiektów" znalazł się uproszczony diagram biblioteki kontenerów Javy. Pora na kompletny diagram biblioteki kontenerów, obejmujący klasy abstrakcyjne i komponenty odziedziczone po poprzednich wersjach Javy (z wyjątkiem implementacji `Queue`):

W Javie SE5 dodano:

- Interfejs `Queue` (implementowany przez `LinkedList`, o czym była już mowa w rozdziale „Kolekcje obiektów") i jego implementacje `PriorityQueue` oraz rozmaite odmiany `BlockingQueue` prezentowane w rozdziale „Współbieżność".
- Interfejs `ConcurrentMap` wraz z implementacją `ConcurrentHashMap` do użytku w środowiskach wielowątkowych; zobacz rozdział „Współbieżność".
- `CopyOnWriteArrayList` i `CopyOnWriteArraySet`, również przygotowane pod kątem wykorzystania współbieżnego.



Pełna taksonomia kontenerów

- `EnumSet` i `EnumMap` — specjalne implementacje `Set` i `Map` przygotowane pod kątem typów wyliczeniowych; zobacz rozdział „Typy wyliczeniowe".
- Zestaw narzędzi w klasie `Collections`.

Prostokąty diagramu rysowane dłuższą przerywaną kreską reprezentują klasy abstrakcyjne — widać zresztą, że zaczynają się od członu `Abstract`. Z początku może to być nieco mylące, ale są one najzwyklejszymi narzędziami częściowo implementującymi dany interfejs. Na przykład gdybyś implementował własny kontener `Set`, nie musiałbyś zaczynać od interfejsu `Set` i samodzielnego implementowania kompletu metod interfejsu — mógłbyś wyprowadzić własny kontener z klasy `AbstractSet`, minimalizując nakłady programistyczne potrzebne do własnej implementacji. Niemniej jednak biblioteka kontenerów jest na tyle wyczerpująca, że spełnia większość typowych potrzeb programistów i rzadko kiedy trzeba definiować własne klasy kontenerów — dlatego klasy `Abstract...` można w większości przypadków śmiało ignorować.

Wypełnianie kontenerów

Mimo że sprawa wypisywania zawartości kontenerów została załatwiona, to wypełnienie kontenerów ma te same wady co `java.util.Arrays`. Podobnie jak w przypadku tablic istnieje `Arrays`, tak dla kontenerów mamy klasę towarzyszącą o nazwie `Collections`, zawierającą statyczne metody usługowe, między nimi metodę o nazwie `fill()`. Ta metoda

również po prostu kopiuje pojedynczą referencję do obiektu do całego kontenera i przy tym działa tylko dla obiektów typu `List`, choć otrzymana lista może być przekazana do konstruktora kontenera innej klasy albo do metody `addAll()`:

```
//: containers/FillingLists.java
// Metody Collections.fill() i Collections.nCopies().
import java.util.*;

class StringAddress {
  private String s;
  public StringAddress(String s) { this.s = s; }
  public String toString() {
    return super.toString() + " " + s;
  }
}

public class FillingLists {
  public static void main(String[] args) {
    List<StringAddress> list= new ArrayList<StringAddress>(
      Collections.nCopies(4, new StringAddress("Ahoj")));
    System.out.println(list);
    Collections.fill(list, new StringAddress("przygodo!"));
    System.out.println(list);
  }
} /* Output: (Sample)
[StringAddress@1f6a7b9 Ahoj, StringAddress@1f6a7b9 Ahoj, StringAddress@1f6a7b9 Ahoj,
StringAddress@1f6a7b9 Ahoj]
[StringAddress@7d772e przygodo!, StringAddress@7d772e przygodo!, StringAddress@7d772e przygodo!,
StringAddress@7d772e przygodo!]
*///:~
```

Przykład pokazuje dwa sposoby wypełniania kolekcji `Collection` referencjami pojedynczego obiektu. Pierwszy sposób, w postaci wywołania `Collections.nCopies()`, tworzy listę `List` przekazywaną do konstruktora; tak o to powstaje wypełniony kontener `ArrayList`.

Metoda `toString()` klasy `StringAddress` wywołuje metodę `Object.toString()`, która z kolei zwraca nazwę klasy uzupełnioną szesnastkową reprezentacją *kodu haszującego* (zwanego też *skrótem*) obiektu (generowanego wywołaniem metody `hashCode()`). Na wyjściu programu widać, że wszystkie referencje odnoszą się do tego samego obiektu; to samo dotyczy również drugiej metody, `Collections.fill()`. Metoda ta jest nawet mniej użyteczna przez to, iż wyłącznie zastępuje elementy, które już są na liście, a nie dodaje nowych.

Rozwiązanie z generatorem

Praktycznie wszystkie podtypy `Collection` dysponują konstruktorem przyjmującym inny egzemplarz `Collection`, będący podstawą do utworzenia nowego kontenera. Aby ułatwić sobie tworzenie danych testowych, musimy więc jedynie oprogramować klasę, która przyjmie argumenty konstruktora klasy `Generator` (zdefiniowanej w rozdziale „Typy ogólne" i wykorzystywanej również w rozdziale „Tablice") oraz dodatkowo liczbę elementów (`quantity`):

```
// : net/mindview/util/CollectionData.java
// Wypełnianie kolekcji Collection danymi z obiektu generatora.
package net.mindview.util;
import java.util.*;

public class CollectionData<T> extends ArrayList<T> {
  public CollectionData(Generator<T> gen, int quantity) {
    for(int i = 0; i < quantity; i++)
      add(gen.next());
  }
  // Uogólniona metoda pomocnicza:
  public static <T> CollectionData<T>
  list(Generator<T> gen, int quantity) {
    return new CollectionData<T>(gen, quantity);
  }
} ///:~
```

W ten sposób za pomocą generatora wypełnimy kontener pożądaną liczbą obiektów. Wynikowy kontener nadaje się do przekazania do konstruktora dowolnego podtypu `Collection` w celu skopiowania elementów do kontenera docelowego. Jeśli kontener docelowy już istnieje, to znaczy został wcześniej skonstruowany, to wygenerowane elementy można do niego skopiować za pomocą metody `addAll()`, również wchodzącej w skład każdego podtypu `Collection`.

Uogólniona metoda pomocnicza z powyższego kodu służy jedynie do zmniejszania ilości kodu towarzyszącego użyciu klasy.

Klasa `CollectionData` to realizacja wzorca projektowego *Adapter*[1]; adaptuje ona generator `Generator` do wymogów konstruktora `Collection`.

Oto przykład inicjalizujący kontener `LinkedHashSet`:

```
// : containers/CollectionDataTest.java
import java.util.*;
import net.mindview.util.*;

class Government implements Generator<String> {
  String[] foundation = ("Dziwne kobiety pływające w stawach " +
    "i rozdające miecze nie mogą decydować o władzy").split(" ");
  private int index;
  public String next() { return foundation[index++]; }
}

public class CollectionDataTest {
  public static void main(String[] args) {
    Set<String> set = new LinkedHashSet<String>(
      new CollectionData<String>(new Government(), 13));
    // Użycie metody pomocniczej:
    set.addAll(CollectionData.list(new Government(), 13));
    System.out.println(set);
  }
} /* Output:
[Dziwne, kobiety, pływające, w, stawach, i, rozdające, miecze, nie, mogą, decydować, o, władzy]
*///:~
```

[1] Być może realizacja nie do końca zgodna ze ścisłą definicją adaptera podaną w książce *Design Patterns*, ale z pewnością zgodna z jej duchem.

Kolejność elementów jest zgodna z kolejnością wstawiania, bo LinkedHashSet utrzymuje wewnętrznie listę reprezentującą pierwotną kolejność elementów.

Dla adaptera CollectionData są teraz dostępne również wszystkie generatory zdefiniowane w rozdziale „Tablice". Oto przykład używający dwóch takich generatorów:

```
//: containers/CollectionDataGeneration.java
// Zastosowanie generatorów z rozdziału "Tablice".
import java.util.*;
import net.mindview.util.*;

public class CollectionDataGeneration {
  public static void main(String[] args) {
    System.out.println(new ArrayList<String>(
      CollectionData.list( // Metoda pomocnicza
        new RandomGenerator.String(9), 10)));
    System.out.println(new HashSet<Integer>(
      new CollectionData<Integer>(
        new RandomGenerator.Integer(), 10)));
  }
} /* Output:
[YNzbrnyGc, FOWZnTcQr, GseGZMmJM, RoEsuEcUO, neOEdLsmw, HLGEahKcx, rEqUCBbkI,
naMesbtWH, kjUrUkZPg, wsqPzDyCy]
[573, 4779, 871, 4367, 6090, 7882, 2017, 8037, 3455, 299]
*///:~
```

Długość ciągów zwracanych przez RandomGenerator.String jest regulowana argumentem wywołania konstruktora.

Generatory dla kontenerów asocjacyjnych

Podobne podejście można zastosować wobec kontenerów asocjacyjnych Map, tyle że wymaga to użycia klasy Pair, bo każde wywołanie metody next() generatora wypełniającego kontener Map musi podawać parę obiektów (jeden klucz i jedną wartość):

```
//: net/mindview/util/Pair.java
package net.mindview.util;

public class Pair<K,V> {
  public final K key;
  public final V value;
  public Pair(K k, V v) {
    key = k;
    value = v;
  }
} ///:~
```

Pola key i value są oznaczone jako publiczne i finalne, więc wynikowa para staje się obiektem tylko do odczytu (wedle koncepcji *Data Transfer Object* mamy tu do czynienia z realizacją wzorca projektowego *Messenger*).

Adapter dla kontenera Map może teraz przy wypełnianiu obiektów inicjalizujących kontenery asocjacyjne korzystać z rozmaitych kombinacji generatorów, implementacji interfejsu Iterable i stałych:

```
//: net/mindview/util/MapData.java
// Kontener Map wypełniany danymi z obiektu generatora.
package net.mindview.util;
import java.util.*;

public class MapData<K,V> extends LinkedHashMap<K,V> {
  // Pojedynczy generator par:
  public MapData(Generator<Pair<K,V>> gen, int quantity) {
    for(int i = 0; i < quantity; i++) {
      Pair<K,V> p = gen.next();
      put(p.key, p.value);
    }
  }
  // Dwa oddzielne generatory:
  public MapData(Generator<K> genK, Generator<V> genV,
      int quantity) {
    for(int i = 0; i < quantity; i++) {
      put(genK.next(), genV.next());
    }
  }
  // Generator kluczy z pojedynczą wartością:
  public MapData(Generator<K> genK, V value, int quantity){
    for(int i = 0; i < quantity; i++) {
      put(genK.next(), value);
    }
  }
  // Iterable i generator wartości:
  public MapData(Iterable<K> genK, Generator<V> genV) {
    for(K key : genK) {
      put(key, genV.next());
    }
  }
  // Iterable i pojedyncza wartość:
  public MapData(Iterable<K> genK, V value) {
    for(K key : genK) {
      put(key, value);
    }
  }
  // Uogólnione metody pomocnicze:
  public static <K,V> MapData<K,V>
  map(Generator<Pair<K,V>> gen, int quantity) {
    return new MapData<K,V>(gen, quantity);
  }
  public static <K,V> MapData<K,V>
  map(Generator<K> genK, Generator<V> genV, int quantity) {
    return new MapData<K,V>(genK, genV, quantity);
  }
  public static <K,V> MapData<K,V>
  map(Generator<K> genK, V value, int quantity) {
    return new MapData<K,V>(genK, value, quantity);
  }
  public static <K,V> MapData<K,V>
  map(Iterable<K> genK, Generator<V> genV) {
    return new MapData<K,V>(genK, genV);
  }
  public static <K,V> MapData<K,V>
  map(Iterable<K> genK, V value) {
    return new MapData<K,V>(genK, value);
  }
} ///:~
```

Możemy teraz wedle uznania korzystać z pojedynczego generatora `Generator<Pair<K, V>>`, dwóch osobnych generatorów, generatora z wartością stałą, implementacji `Iterable` (a więc również dowolnej kolekcji) z generatorem czy `Iterable` z wartością stałą. Ilość kodu potrzebnego do tworzenia obiektu `MapData` zmniejszają wygodne metody pomocnicze.

Pora na przykład zastosowania obiektu `MapData`. Generator `Letters` implementuje również interfejs `Iterable`, zwracając `Iterator`; dzięki temu przetestujemy metody `MapData.map()` pod kątem obsługi obiektów `Iterable`:

```
// containers/MapDataTest.java
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class Letters implements Generator<Pair<Integer,String>>,
  Iterable<Integer> {
  private int size = 9;
  private int number = 1;
  private char letter = 'A';
  public Pair<Integer,String> next() {
    return new Pair<Integer,String>(
      number++, "" + letter++);
  }
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      public Integer next() { return number++; }
      public boolean hasNext() { return number < size; }
      public void remove() {
        throw new UnsupportedOperationException();
      }
    };
  }
}

public class MapDataTest {
  public static void main(String[] args) {
    // Generator par:
    print(MapData.map(new Letters(), 11));
    // Para generatorów:
    print(MapData.map(new CountingGenerator.Character(),
      new RandomGenerator.String(3), 8));
    // Generator kluczy z pojedynczą wartością:
    print(MapData.map(new CountingGenerator.Character(),
      "Wartość", 6));
    // Iterable z generatorem wartości:
    print(MapData.map(new Letters(),
      new RandomGenerator.String(3)));
    // Iterable z pojedynczą wartością:
    print(MapData.map(new Letters(), "Test"));
  }
} /* Output:
{1=A, 2=B, 3=C, 4=D, 5=E, 6=F, 7=G, 8=H, 9=I, 10=J, 11=K}
{a=YNz, b=brn, c=yGc, d=FOW, e=ZnT, f=cQr, g=Gse, h=GZM}
{a=Wartość, b=Wartość, c=Wartość, d=Wartość, e=Wartość, f=Wartość}
{1=mJM, 2=RoE, 3=suE, 4=cUO, 5=neO, 6=EdL, 7=smw, 8=HLG}
{1=Test, 2=Test, 3=Test, 4=Test, 5=Test, 6=Test, 7=Test, 8=Test}
*///:~
```

W przykładzie wykorzystaliśmy generatory z rozdziału „Tablice".

Za pomocą przedstawionych narzędzi można tworzyć dowolne generowane zbiory danych dla odwzorowań i kolekcji, a potem inicjalizować te odwzorowania i kolekcje za pomocą konstruktora albo metod `Map.putAll()` czy `Collection.addAll()`.

Stosowanie klas abstrakcyjnych

Problem generowania danych testowych do wypełniania kontenerów można zaatakować również inaczej, tworząc własne implementacje `Collection` i `Map`. Każdy kontener `java.util` posiada własną klasę abstrakcyjną udostępniającą częściową implementację interfejsu tego kontenera, którą można wykorzystać jako podstawę własnej implementacji. Jeśli wynikowy kontener ma być kontenerem tylko do odczytu, jak to zazwyczaj ma miejsce w przypadku danych testowych, liczba metod, które trzeba oprogramować samodzielnie, jest już całkiem mała.

Choć w naszym przypadku nie jest to szczególnie potrzebne, zaprezentowane rozwiązanie daje też okazję do zilustrowania kolejnego wzorca projektowego — *Flyweight* („waga musza"). Wykorzystuje się go tam, gdzie zwykłe rozwiązanie angażuje zbyt wiele obiektów albo gdy tworzenie normalnych obiektów zajmuje zbyt dużo miejsca. Wzorzec projektowy Flyweight wyodrębnia części obiektu na zewnątrz tak, aby jakaś część integralnych składowych obiektu, zamiast znajdować się wprost w obiekcie, znajdowała się w jakiejś wydajnej zewnętrznej tabeli (albo była generowana na bieżąco w razie potrzeby).

Ważnym elementem tego przykładu jest ilustracja prostoty utworzenia własnego kontenera asocjacyjnego `Map` i własnej kolekcji `Collection` w wyniku dziedziczenia po klasach abstrakcyjnych. Aby powołać do życia własny kontener `Map`, wystarczy wyprowadzić klasę z `AbstractMap` i (jeśli kontener ma być tylko do odczytu) zdefiniować metodę `entrySet()`. Aby utworzyć własny kontener `Set` tylko do odczytu, wystarczy wyprowadzić klasę z `AbstractSet` i zaimplementować metody `iterator()` oraz `size()`.

Zbiór danych z tego przykładu to kontener asocjacyjny kojarzący nazwy krajów ze stolicami[2]. Metoda `capitals()` zwraca kontener `Map` krajów i stolic. Metoda `names()` zwraca listę (`List`) samych nazw krajów. W obu przypadkach można ograniczyć wynik do podzbioru krajów — wystarczy przekazać argument określający pożądany rozmiar listy (odwzorowania):

```
// : net/mindview/util/Countries.java
// Kontenery Map i List danych testowych "w wadze muszej".
package net.mindview.util;
import java.util.*;
import static net.mindview.util.Print.*;

public class Countries {
  public static final String[][] DATA = {
    // Afryka
    {"ALGERIA","Algiers"}, {"ANGOLA","Luanda"},
    {"BENIN","Porto-Novo"}, {"BOTSWANA","Gaberone"},
    {"BULGARIA","Sofia"}, {"BURKINA FASO","Ouagadougou"},
```

[2] Pierwotne dane znalazłem w Internecie; w miarę upływu czasu Czytelnicy zgłaszali liczne poprawki.

```
{"BURUNDI","Bujumbura"},
{"CAMEROON","Yaounde"}, {"CAPE VERDE","Praia"},
{"CENTRAL AFRICAN REPUBLIC","Bangui"},
{"CHAD","N'djamena"}, {"COMOROS","Moroni"},
{"CONGO","Brazzaville"}, {"DJIBOUTI","Dijibouti"},
{"EGYPT","Cairo"}, {"EQUATORIAL GUINEA","Malabo"},
{"ERITREA","Asmara"}, {"ETHIOPIA","Addis Ababa"},
{"GABON","Libreville"}, {"THE GAMBIA","Banjul"},
{"GHANA","Accra"}, {"GUINEA","Conakry"},
{"BISSAU","Bissau"},
{"COTE D'IVOIR (IVORY COAST)","Yamoussoukro"},
{"KENYA","Nairobi"}, {"LESOTHO","Maseru"},
{"LIBERIA","Monrovia"}, {"LIBYA","Tripoli"},
{"MADAGASCAR","Antananarivo"}, {"MALAWI","Lilongwe"},
{"MALI","Bamako"}, {"MAURITANIA","Nouakchott"},
{"MAURITIUS","Port Louis"}, {"MOROCCO","Rabat"},
{"MOZAMBIQUE","Maputo"}, {"NAMIBIA","Windhoek"},
{"NIGER","Niamey"}, {"NIGERIA","Abuja"},
{"RWANDA","Kigali"},
{"SAO TOME E PRINCIPE","Sao Tome"},
{"SENEGAL","Dakar"}, {"SEYCHELLES","Victoria"},
{"SIERRA LEONE","Freetown"}, {"SOMALIA","Mogadishu"},
{"SOUTH AFRICA","Pretoria/Cape Town"},
{"SUDAN","Khartoum"},
{"SWAZILAND","Mbabane"}, {"TANZANIA","Dodoma"},
{"TOGO","Lome"}, {"TUNISIA","Tunis"},
{"UGANDA","Kampala"},
{"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)",
 "Kinshasa"},
{"ZAMBIA","Lusaka"}, {"ZIMBABWE","Harare"},
// Azja
{"AFGHANISTAN","Kabul"}, {"BAHRAIN","Manama"},
{"BANGLADESH","Dhaka"}, {"BHUTAN","Thimphu"},
{"BRUNEI","Bandar Seri Begawan"},
{"CAMBODIA","Phnom Penh"},
{"CHINA","Beijing"}, {"CYPRUS","Nicosia"},
{"INDIA","New Delhi"}, {"INDONESIA","Jakarta"},
{"IRAN","Tehran"}, {"IRAQ","Baghdad"},
{"ISRAEL","Jerusalem"}, {"JAPAN","Tokyo"},
{"JORDAN","Amman"}, {"KUWAIT","Kuwait City"},
{"LAOS","Vientiane"}, {"LEBANON","Beirut"},
{"MALAYSIA","Kuala Lumpur"}, {"THE MALDIVES","Male"},
{"MONGOLIA","Ulan Bator"},
{"MYANMAR (BURMA)","Rangoon"},
{"NEPAL","Katmandu"}, {"NORTH KOREA","P'yongyang"},
{"OMAN","Muscat"}, {"PAKISTAN","Islamabad"},
{"PHILIPPINES","Manila"}, {"QATAR","Doha"},
{"SAUDI ARABIA","Riyadh"}, {"SINGAPORE","Singapore"},
{"SOUTH KOREA","Seoul"}, {"SRI LANKA","Colombo"},
{"SYRIA","Damascus"},
{"TAIWAN (REPUBLIC OF CHINA)","Taipei"},
{"THAILAND","Bangkok"}, {"TURKEY","Ankara"},
{"UNITED ARAB EMIRATES","Abu Dhabi"},
{"VIETNAM","Hanoi"}, {"YEMEN","Sana'a"},
// Australia i Oceania
{"AUSTRALIA","Canberra"}, {"FIJI","Suva"},
{"KIRIBATI","Bairiki"},
{"MARSHALL ISLANDS","Dalap-Uliga-Darrit"},
```

```
{"MICRONESIA","Palikir"}, {"NAURU","Yaren"},
{"NEW ZEALAND","Wellington"}, {"PALAU","Koror"},
{"PAPUA NEW GUINEA","Port Moresby"},
{"SOLOMON ISLANDS","Honaira"}, {"TONGA","Nuku'alofa"},
{"TUVALU","Fongafale"}, {"VANUATU","< Port-Vila"},
{"WESTERN SAMOA","Apia"},
// Europa Wschodnia i byłe ZSRR
{"ARMENIA","Yerevan"}, {"AZERBAIJAN","Baku"},
{"BELARUS (BYELORUSSIA)","Minsk"},
{"GEORGIA","Tbilisi"},
{"KAZAKSTAN","Almaty"}, {"KYRGYZSTAN","Alma-Ata"},
{"MOLDOVA","Chisinau"}, {"RUSSIA","Moscow"},
{"TAJIKISTAN","Dushanbe"}, {"TURKMENISTAN","Ashkabad"},
{"UKRAINE","Kyiv"}, {"UZBEKISTAN","Tashkent"},
// Europa Zachodnia
{"ALBANIA","Tirana"}, {"ANDORRA","Andorra la Vella"},
{"AUSTRIA","Vienna"}, {"BELGIUM","Brussels"},
{"BOSNIA","-"}, {"HERZEGOVINA","Sarajevo"},
{"CROATIA","Zagreb"}, {"CZECH REPUBLIC","Prague"},
{"DENMARK","Copenhagen"}, {"ESTONIA","Tallinn"},
{"FINLAND","Helsinki"}, {"FRANCE","Paris"},
{"GERMANY","Berlin"}, {"GREECE","Athens"},
{"HUNGARY","Budapest"}, {"ICELAND","Reykjavik"},
{"IRELAND","Dublin"}, {"ITALY","Rome"},
{"LATVIA","Riga"}, {"LIECHTENSTEIN","Vaduz"},
{"LITHUANIA","Vilnius"}, {"LUXEMBOURG","Luxembourg"},
{"MACEDONIA","Skopje"}, {"MALTA","Valletta"},
{"MONACO","Monaco"}, {"MONTENEGRO","Podgorica"},
{"THE NETHERLANDS","Amsterdam"}, {"NORWAY","Oslo"},
{"POLAND","Warsaw"}, {"PORTUGAL","Lisbon"},
{"ROMANIA","Bucharest"}, {"SAN MARINO","San Marino"},
{"SERBIA","Belgrade"}, {"SLOVAKIA","Bratislava"},
{"SLOVENIA","Ljuijana"}, {"SPAIN","Madrid"},
{"SWEDEN","Stockholm"}, {"SWITZERLAND","Berne"},
{"UNITED KINGDOM","London"}, {"VATICAN CITY","---"},
// Ameryka Północna i Środkowa
{"ANTIGUA AND BARBUDA","Saint John's"},
{"BAHAMAS","Nassau"},
{"BARBADOS","Bridgetown"}, {"BELIZE","Belmopan"},
{"CANADA","Ottawa"}, {"COSTA RICA","San Jose"},
{"CUBA","Havana"}, {"DOMINICA","Roseau"},
{"DOMINICAN REPUBLIC","Santo Domingo"},
{"EL SALVADOR","San Salvador"},
{"GRENADA","Saint George's"},
{"GUATEMALA","Guatemala City"},
{"HAITI","Port-au-Prince"},
{"HONDURAS","Tegucigalpa"}, {"JAMAICA","Kingston"},
{"MEXICO","Mexico City"}, {"NICARAGUA","Managua"},
{"PANAMA","Panama City"}, {"ST. KITTS","-"},
{"NEVIS","Basseterre"}, {"ST. LUCIA","Castries"},
{"ST. VINCENT AND THE GRENADINES","Kingstown"},
{"UNITED STATES OF AMERICA","Washington, D.C."},
// Ameryka Południowa
{"ARGENTINA","Buenos Aires"},
{"BOLIVIA","Sucre (legal)/La Paz(administrative)"},
{"BRAZIL","Brasilia"}, {"CHILE","Santiago"},
{"COLOMBIA","Bogota"}, {"ECUADOR","Quito"},
{"GUYANA","Georgetown"}, {"PARAGUAY","Asuncion"},
```

```
    {"PERU","Lima"}, {"SURINAME","Paramaribo"},
    {"TRINIDAD AND TOBAGO","Port of Spain"},
    {"URUGUAY","Montevideo"}, {"VENEZUELA","Caracas"},
  };
  // Zastosowanie klasy AbstractMap z implementacją metody entrySet()
  private static class FlyweightMap
  extends AbstractMap<String,String> {
    private static class Entry
    implements Map.Entry<String,String> {
      int index;
      Entry(int index) { this.index = index; }
      public boolean equals(Object o) {
        return DATA[index][0].equals(o);
      }
      public String getKey() { return DATA[index][0]; }
      public String getValue() { return DATA[index][1]; }
      public String setValue(String value) {
        throw new UnsupportedOperationException();
      }
      public int hashCode() {
        return DATA[index][0].hashCode();
      }
    }
    // Zastosowanie klasy AbstractSet z implementacjami size() i iterator()
    static class EntrySet
    extends AbstractSet<Map.Entry<String,String>> {
      private int size;
      EntrySet(int size) {
        if(size < 0)
          this.size = 0;
        // Nie może być większy od tablicy:
        else if(size > DATA.length)
          this.size = DATA.length;
        else
          this.size = size;
      }
      public int size() { return size; }
      private class Iter
      implements Iterator<Map.Entry<String,String>> {
        // Tylko jeden obiekt Entry na Iterator:
        private Entry entry = new Entry(-1);
        public boolean hasNext() {
          return entry.index < size - 1;
        }
        public Map.Entry<String,String> next() {
          entry.index++;
          return entry;
        }
        public void remove() {
          throw new UnsupportedOperationException();
        }
      }
      public
      Iterator<Map.Entry<String,String>> iterator() {
        return new Iter();
      }
    }
    private static Set<Map.Entry<String,String>> entries =
```

```
      new EntrySet(DATA.length);
    public Set<Map.Entry<String,String>> entrySet() {
      return entries;
    }
  }
  // Utworzenie częściowego odwzorowania 'size' stolic krajów:
  static Map<String,String> select(final int size) {
    return new FlyweightMap() {
      public Set<Map.Entry<String,String>> entrySet() {
        return new EntrySet(size);
      }
    };
  }
  static Map<String,String> map = new FlyweightMap();
  public static Map<String,String> capitals() {
    return map; // Całe odwzorowanie
  }
  public static Map<String,String> capitals(int size) {
    return select(size); // Odwzorowanie częściowe
  }
  static List<String> names =
    new ArrayList<String>(map.keySet());
  // Komplet nazw krajów:
  public static List<String> names() { return names; }
  // Lista częściowa:
  public static List<String> names(int size) {
    return new ArrayList<String>(select(size).keySet());
  }
  public static void main(String[] args) {
    print(capitals(10));
    print(names(10));
    print(new HashMap<String,String>(capitals(3)));
    print(new LinkedHashMap<String,String>(capitals(3)));
    print(new TreeMap<String,String>(capitals(3)));
    print(new Hashtable<String,String>(capitals(3)));
    print(new HashSet<String>(names(6)));
    print(new LinkedHashSet<String>(names(6)));
    print(new TreeSet<String>(names(6)));
    print(new ArrayList<String>(names(6)));
    print(new LinkedList<String>(names(6)));
    print(capitals().get("BRAZIL"));
  }
} /* Output:
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo, BOTSWANA=Gaberone,
BULGARIA=Sofia, BURKINA FASO=Ouagadougou, BURUNDI=Bujumbura, CAMEROON=Yaounde,
CAPE VERDE=Praia, CENTRAL AFRICAN REPUBLIC=Bangui}
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, BURUNDI, CAMEROON,
CAPE VERDE, CENTRAL AFRICAN REPUBLIC]
{BENIN=Porto-Novo, ANGOLA=Luanda, ALGERIA=Algiers}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
[BULGARIA, BURKINA FASO, BOTSWANA, BENIN, ANGOLA, ALGERIA]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
Brasilia
*///:~
```

Dwuwymiarowa tablica ciągów znaków DATA jest publiczna, więc może być wykorzystywana również gdzie indziej. Klasa FlyweightMap musi implementować metodę entrySet() wymagającą zarówno własnej implementacji Set, jak i własnej klasy Map.Entry. I tu mamy pierwszy przejaw „wagi muszej": każdy obiekt Map.Entry zamiast wartości i klucza przechowuje jedynie indeks. Obsługa wywołania getKey() bądź getValue() polega na wykorzystaniu indeksu do odwołania się do odpowiedniego elementu tablicy DATA. Klasa EntrySet musi gwarantować, że jej rozmiar (size) nie będzie większy od rozmiaru tablicy DATA.

Druga część „wagi muszej" przejawia się w implementacji EntrySet.Iterator. Zamiast osobnego obiektu Map.Entry dla każdej pary danych z tablicy DATA mamy tu tylko po jednym obiekcie Map.Entry dla każdego *iteratora*. Zwracany w tej roli obiekt Entry pełni rolę okna danych; zawiera jedynie indeks do statycznej tablicy ciągów znaków. Każde wywołanie metody next() na rzecz iteratora oznacza inkrementację indeksu w Entry tak, aby indeks odnosił się do kolejnej pary elementów; następnie next() zwraca odpowiedni obiekt Entry[3].

Metoda select() tworzy i zwraca kontener FlyweightMap zawierający zbiór EntrySet o pożądanym rozmiarze; wykorzystujemy to w przeciążonych wersjach metod capitals() i names() testowanych w metodzie main().

W niektórych testach ograniczony rozmiar klasy Countries może stanowić problem. Możemy jednak wykorzystać to samo podejście do tworzenia zainicjalizowanych kontenerów ze zbiorami danych o dowolnych rozmiarach. Poniższa klasa to lista (List) o dowolnym rozmiarze zainicjalizowana (przynajmniej z punktu widzenia klienta) danymi typu Integer:

```
//: net/mindview/util/CountingIntegerList.java
// Lista dowolnego rozmiaru z danymi testowymi.
package net.mindview.util;
import java.util.*;

public class CountingIntegerList
extends AbstractList<Integer> {
  private int size;
  public CountingIntegerList(int size) {
    this.size = size < 0 ? 0 : size;
  }
  public Integer get(int index) {
    return Integer.valueOf(index);
  }
  public int size() { return size; }
  public static void main(String[] args) {
    System.out.println(new CountingIntegerList(30));
  }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
*///:~
```

[3] Kontenery Map z biblioteki java.util wykonują większość operacji kopiowania elementów za pośrednictwem wywołań metod getKey() i getValue(), więc takie podejście będzie skuteczne. Problem pojawiłby się, gdyby własny kontener Map miał kopiować całość Map.Entry.

Aby utworzyć własną listę (tylko do odczytu na podstawie klasy AbstractList), należy zaimplementować metody get() i size(). Znów wykorzystujemy wzorzec „wagi muszej": metoda get() tworzy wartość na żądanie, więc sama lista nie musi być faktycznie wypełniona danymi.

Oto kontener Map „zainicjalizowany" parami Integer-String; on też może mieć dowolny rozmiar:

```
//: net/mindview/util/CountingMapData.java
// Niewyczerpywalny kontener Map z danymi testowymi.
package net.mindview.util;
import java.util.*;

public class CountingMapData
extends AbstractMap<Integer,String> {
  private int size;
  private static String[] chars =
    "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z"
    .split(" ");
  public CountingMapData(int size) {
    if(size < 0) this.size = 0;
    this.size = size;
  }
  private static class Entry
  implements Map.Entry<Integer,String> {
    int index;
    Entry(int index) { this.index = index; }
    public boolean equals(Object o) {
      return Integer.valueOf(index).equals(o);
    }
    public Integer getKey() { return index; }
    public String getValue() {
      return
        chars[index % chars.length] +
        Integer.toString(index / chars.length);
    }
    public String setValue(String value) {
      throw new UnsupportedOperationException();
    }
    public int hashCode() {
      return Integer.valueOf(index).hashCode();
    }
  }
  public Set<Map.Entry<Integer,String>> entrySet() {
    // Kontener LinkedHashSet zachowuje porządek elementów:
    Set<Map.Entry<Integer,String>> entries =
      new LinkedHashSet<Map.Entry<Integer,String>>();
    for(int i = 0; i < size; i++)
      entries.add(new Entry(i));
    return entries;
  }
  public static void main(String[] args) {
    System.out.println(new CountingMapData(60));
  }
} /* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0, 10=K0, 11=L0, 12=M0, 13=N0,
14=O0, 15=P0, 16=Q0, 17=R0, 18=S0, 19=T0, 20=U0, 21=V0, 22=W0, 23=X0, 24=Y0, 25=Z0, 26=A1,
```

```
27=B1, 28=C1, 29=D1, 30=E1, 31=F1, 32=G1, 33=H1, 34=I1, 35=J1, 36=K1, 37=L1, 38=M1, 39=N1,
40=O1, 41=P1, 42=Q1, 43=R1, 44=S1, 45=T1, 46=U1, 47=V1, 48=W1, 49=X1, 50=Y1, 51=Z1, 52=A2,
53=B2, 54=C2, 55=D2, 56=E2, 57=F2, 58=G2, 59=H2}
*///:~
```

Tym razem zamiast tworzyć własną klasę Set, skorzystaliśmy z kontenera LinkedHashSet, więc sprzeniewierzyliśmy się częściowo „wadze muszej".

Ćwiczenie 1. Utwórz listę List (wypróbuj zarówno ArrayList, jak i LinkedList) i wypełnij ją za pomocą klasy Countries. Posortuj listę i wypisz ją na wyjściu programu, a następnie kilkukrotnie zastosuj do listy metodę Collections.shuffle(), za każdym razem wypisując wynik i sprawdzając, czy każde wywołanie tasuje listę inaczej niż poprzednie (1).

Ćwiczenie 2. Wygeneruj kontenery Set i Map zawierające wszystkie kraje, których nazwy zaczynają się od litery 'A' (2).

Ćwiczenie 3. Za pomocą klasy Countries wypełnij kilkukrotnie kontener Set tymi samymi danymi i sprawdź, czy ostatecznie w zbiorze faktycznie nie ma żadnych duplikatów elementów. Sprawdź to dla kontenerów HashSet, LinkedHashSet i TreeSet (1).

Ćwiczenie 4. Utwórz inicjalizator Collection, który za pomocą klasy TextFile otworzy plik i wyodrębni z niego poszczególne słowa, a następnie wykorzysta je w roli źródła danych dla wynikowej kolekcji. Sprawdź całość w działaniu (2).

Ćwiczenie 5. Zmodyfikuj program *CountingMapData.java* pod kątem pełnej implementacji wzorca „wagi muszej"; ma to polegać na dodaniu własnej klasy EntrySet, jak w programie *Countries.java* (3).

Interfejs Collection

Poniższa tabela pokazuje wszystko, co można zrobić z Collection (nie uwzględniając metod automatycznie odziedziczonych po klasie Object), a zatem również wszystko, co można zrobić ze zbiorami Set lub listami (List ma jeszcze dodatkowe funkcje). Natomiast odwzorowania Map nie wywodzą się od Collection, toteż rozpatrzymy je osobno.

Metoda	Działanie
`boolean add(T)`	Zapewnia, że kontener zawiera argument typu uogólnionego T. Zwraca wartość false, jeśli nie dodaje argumentu (jest to metoda opcjonalna, opisana dalej w tym rozdziale).
`boolean addAll(Collection <? extends T>)`	Umieszcza wszystkie elementy z konteneru-parametru w docelowej kolekcji. Zwraca wartość true, jeżeli dodano przynajmniej jeden element (jest opcjonalna).
`void clear()`	Usuwa wszystkie elementy z kontenera (jest opcjonalna).
`boolean contains(T)`	true, jeśli kontener zawiera argument typu uogólnionego T.
`boolean containsAll(Collection<?>)`	true, gdy kontener zawiera wszystkie z elementów zawartych w argumencie.
`boolean isEmpty()`	true, jeśli kontener jest pusty.

Metoda	Działanie
`Iterator<T> iterator()`	Zwraca `Iterator<T>`, którego można następnie użyć do przechodzenia po elementach kontenera.
`boolean remove(Object)`	Jeśli parametr jest w kontenerze, to jeden z jego egzemplarzy zostanie usunięty. Zwraca wartość `true`, jeśli doszło do usunięcia (jest opcjonalna).
`boolean removeAll(Collection<?>)`	Usuwa wszystkie elementy kontenera, znajdujące się również w przekazanym metodzie kontenerze-argumencie. Zwraca wartość `true`, jeśli jakiekolwiek usunięcie miało miejsce (jest opcjonalna).
`boolean retainAll(Collection<?>)`	Pozostawia w kontenerze wyłącznie elementy, które są także zawarte w argumencie („część wspólna" w teorii zbiorów). Zwraca wartość `true`, jeśli miały miejsce jakieś zmiany (jest opcjonalna).
`int size()`	Zwraca liczbę elementów zawartych w kontenerze.
`Object[] toArray()`	Zwraca tablicę zawierającą wszystkie elementy z kontenera.
`<T> T[] toArray(T[] a)`	Zwraca tablicę zawierającą wszystkie elementy kontenera. Dynamiczny typ tablicy wynikowej to nie `Object` — typ jest zgodny z typem tablicy podanej w roli argumentu `a`.

Jak widać, nie ma funkcji `get()` pozwalającej na swobodny wybór elementów. Jest to spowodowane tym, że do kategorii `Collection` zaliczają się również zbiory `Set`, które utrzymują własny wewnętrzny porządek (zatem czynią dostęp swobodny bezsensownym). Tak więc jeśli chcemy obejrzeć wszystkie elementy kontenera `Collection`, to trzeba użyć iteratora; jest to jedyny sposób na odzyskanie elementów.

Następny przykład pokazuje wszystkie wymienione metody. Powtarzam, działa to wobec wszystkiego, co dziedziczy z `Collection`, ale stosuję tu `ArrayList` jako „najmniejszy wspólny mianownik":

```
// : containers/CollectionMethods.java
// Co można zrobić z każdym podtypem Collections.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class CollectionMethods {
  public static void main(String[] args) {
    Collection<String> c = new ArrayList<String>();
    c.addAll(Countries.names(6));
    c.add("dziesięć");
    c.add("jedenaście");
    print(c);
    // Utworzenie tablicy z kopią elementów listy:
    Object[] array = c.toArray();
    // Utworzenie tablicy ciągów znaków na podstawie listy:
    String[] str = c.toArray(new String[0]);
    // Wyszukanie elementu największego i najmniejszego;
    // znaczenie tych operacji bywa różne, zależnie od sposobu
    // implementowania interfejsu Comparable dla elementów:
    print("Collections.max(c) = " + Collections.max(c));
    print("Collections.min(c) = " + Collections.min(c));
    // Dodawanie kolekcji do innych kolekcji
    Collection<String> c2 = new ArrayList<String>();
    c2.addAll(Countries.names(6));
```

```
    c.addAll(c2);
    print(c);
    c.remove(Countries.DATA[0][0]);
    print(c);
    c.remove(Countries.DATA[1][0]);
    print(c);
    // Usunięcie wszyskich elementów wymienionych
    // w kolekcji przekazanej argumentem wywołania:
    c.removeAll(c2);
    print(c);
    c.addAll(c2);
    print(c);
    // Czy w tej kolekcji znajduje się dany element?
    String val = Countries.DATA[3][0];
    print("c.contains(" + val  + ") = " + c.contains(val));
    // Czy w tej kolekcji znajduje się komplet elementów danej kolekcji?
    print("c.containsAll(c2) = " + c.containsAll(c2));
    Collection<String> c3 =
      ((List<String>)c).subList(3, 5);
    // Zachowanie wszystkich elementów znajdujących się w
    // kolekcjach c2 i c3 (część wspólna kolekcji):
    c2.retainAll(c3);
    print(c2);
    // Pozbycie się elementów z c2, które znajdują się
    // również w kolekcji c3:
    c2.removeAll(c3);
    print("c2.isEmpty() = " +  c2.isEmpty());
    c = new ArrayList<String>();
    c.addAll(Countries.names(6));
    print(c);
    c.clear(); // Usunięcie wszystkich elementów kolekcji
    print("po c.clear():" + c);
  }
} /* Output:
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, dziesięć, jedenaście]
Collections.max(c) = dziesięć
Collections.min(c) = ALGERIA
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, dziesięć, jedenaście,
ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, dziesięć, jedenaście, ALGERIA,
ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[BENIN, BOTSWANA, BULGARIA, BURKINA FASO, dziesięć, jedenaście, ALGERIA, ANGOLA, BENIN,
BOTSWANA, BULGARIA, BURKINA FASO]
[dziesięć, jedenaście]
[dziesięć, jedenaście, ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
c.contains(BOTSWANA) = true
c.containsAll(c2) = true
[ANGOLA, BENIN]
c2.isEmpty() = true
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
po c.clear():[]
*///:~
```

Listy klasy `ArrayList` są tworzone z różną zawartością i rzutowane w górę do `Collection`, tak więc oczywiste jest, że nic innego poza funkcjami interfejsu `Collection` nie jest tutaj używane. W metodzie `main()` na prostych przykładach można prześledzić wszystkie metody interfejsu `Collection`.

Następne podrozdziały opisują różne implementacje interfejsów: `List`, `Set` i `Map` oraz w każdym przypadku sygnalizują (znakiem gwiazdki), która z nich powinna być domyślnie stosowana. Omówienie starszych klas: `Vector`, `Stack` i `Hashtable`, zostało odłożone na koniec rozdziału; choć nie powinno się już ich używać, wciąż można je znaleźć w starszym kodzie.

Operacje opcjonalne

Interfejs `Collection` przewiduje zestaw *metod opcjonalnych* służących do dodawania i usuwania elementów kolekcji. Opcjonalność oznacza, że klasa implementująca interfejs nie ma obowiązku udostępniać definicji tych metod.

To dość niezwykły sposób definiowania interfejsu. Dotychczas traktowaliśmy interfejs w projektowaniu obiektowym jako rodzaj umowy: „nie jest ważne, jak interfejs będzie implementowany, byle do obiektu implementującego interfejs dawały się wysyłać zdefiniowane w nim komunikaty"[4]. Ale opcjonalność operacji interfejsu narusza tę fundamentalną zasadę; wynika z niej bowiem, że wywołanie niektórych metod nie da oczekiwanych albo jakichkolwiek sensownych efektów. Ba, wywołanie niezdefiniowanych operacji opcjonalnych doprowadzi do zgłoszenia wyjątku! Czyżbyśmy odrzucali statyczną kontrolę typów?

Nie jest aż tak źle. Jeśli nawet operacja jest opcjonalna, kompilator nadal ograniczy nas do wywoływania jedynie metod interfejsu, a więc nie tak, jak w językach dynamicznych, gdzie można wywołać dowolną metodę dowolnego obiektu i przekonać się, dopiero po uruchomieniu programu, czy to wywołanie miało jakikolwiek efekt[5]. W dodatku większość metod, które pobierają odwołanie do `Collection` jako parametr, jedynie *czyta* z takiej kolekcji — zaś żadne z metod „czytających" `Collection` *nie* są opcjonalne.

Po cóż nam więc metody opcjonalne? Otóż takie rozwiązanie zapobiega eksplozji liczby interfejsów podczas projektowania. Inne rozwiązania bibliotek kontenerowych zawsze zdają się posiadać nadmierną liczbę interfejsów do opisu każdej z odmian głównego schematu. Zresztą nie można wyodrębnić wszystkich przypadków specjalnych w interfejsach, ponieważ ktoś może zawsze wymyślić jakiś nowy interfejs. Rozwiązanie z oznaczeniem operacji jako „nieobsługiwanych" realizuje ważny cel biblioteki kontenerowej Javy: kontenery są proste w nauce i użyciu; operacje nieobsługiwane są specjalnym przypadkiem, którym możemy zająć się później. Jednak do funkcjonowania tego mechanizmu:

1. Wyjątek `UnsupportedOperationException` musi być zdarzeniem zachodzącym rzadko. Oznacza to, że dla większości klas wszystkie operacje powinny działać i tylko w szczególnych przypadkach operacja może nie być obsługiwana. Jest to aktualne w bibliotece kontenerowej Javy, gdyż klasy, które będziesz wykorzystywał przez 99 procent czasu — `ArrayList`, `LinkedList`, `HashSet` i `HashMap`, podobnie jak inne konkretne implementacje — obsługują wszystkie z operacji. Projekt biblioteki

[4] Słowo interfejs występuje tu zarówno w znaczeniu formalnego słowa kluczowego `interface`, jak i w znaczeniu ogólniejszym, oznaczającym „metody obsługiwane przez klasę i jej podklasy".

[5] W takim ujęciu wydaje się to dziwaczne i bezużyteczne, ale w rozdziale „Informacje o typach" przekonaliśmy się, że tego rodzaju dynamiczne zachowanie może być bardzo pomocne.

zapewnia „tylne drzwi", kiedy chcemy stworzyć nową implementację `Collection` bez dostarczania definicji dla wszystkich metod tego interfejsu i przy tym nadal pasującą do istniejącej biblioteki.

2. Jeżeli już operacja *jest* nieobsługiwana, to powinno istnieć wszelkie prawdopodobieństwo, że wyjątek `UnsupportedOperationException` wystąpi w czasie implementacji, a nie po dostarczeniu produktu do klienta. W końcu sygnalizuje błąd programowania: implementację zastosowano niepoprawnie. Ten punkt jest mniej pewny i tu właśnie wchodzą w grę doświadczenie i eksperymenty. Dopiero po pewnym czasie odkryjemy, jak dobrze to działa.

Warto zaznaczyć, że nieobsługiwane operacje dają się wykryć dopiero w czasie wykonania, więc reprezentują element dynamicznej kontroli typów. Programiści przywykli do języków programowania ze statyczną kontrolą typów, jak C++, mogą uznać Javę za kolejny taki język. Otóż Java niewątpliwie *dysponuje* statyczną kontrolą typów, ale posiada również znaczny udział kontroli dynamicznej, więc trudno zakwalifikować ją do konkretnej kategorii typowania (w całości dynamicznego lub wyłącznie statycznego). Kiedy zaczniesz to zauważać, dostrzeżesz również inne przykłady dynamicznej kontroli typów w Javie.

Operacje nieobsługiwane

Kwestia operacji nieobsługiwanych pojawia się typowo przy kontenerach, które w roli pamięci wykorzystują struktury danych o stałych rozmiarach. Kontener taki otrzymujemy choćby w wyniku zamiany tablicy w kontener `List` za pomocą wywołania `Arrays.asList()`. Można też *zdecydować*, aby dowolny kontener (również kontener `Map`) zgłaszał wyjątek `UnsupportedOperationException`, korzystając z „niemodyfikowalnych" metod klasy `Collections`. Poniższy przykład ilustruje oba przypadki:

```
//: containers/Unsupported.java
// Nieobsługiwane operacje kontenerów Javy.
import java.util.*;

public class Unsupported {
  static void test(String msg, List<String> list) {
    System.out.println("--- " + msg + " ---");
    Collection<String> c = list;
    Collection<String> subList = list.subList(1,8);
    // Kopiowanie fragmentu listy:
    Collection<String> c2 = new ArrayList<String>(subList);
    try { c.retainAll(c2); } catch(Exception e) {
      System.out.println("retainAll(): " + e);
    }
    try { c.removeAll(c2); } catch(Exception e) {
      System.out.println("removeAll(): " + e);
    }
    try { c.clear(); } catch(Exception e) {
      System.out.println("clear(): " + e);
    }
    try { c.add("X"); } catch(Exception e) {
      System.out.println("add(): " + e);
    }
    try { c.addAll(c2); } catch(Exception e) {
      System.out.println("addAll(): " + e);
```

```
    }
    try { c.remove("C"); } catch(Exception e) {
      System.out.println("remove(): " + e);
    }
    // Metoda List.set() modyfikuje wartość, ale
    // nie zmienia rozmiaru struktury danych:
    try {
      list.set(0, "X");
    } catch(Exception e) {
      System.out.println("List.set(): " + e);
    }
  }
  public static void main(String[] args) {
    List<String> list =
      Arrays.asList("A B C D E F G H I J K L".split(" "));
    test("Modyfikowalna kopia", new ArrayList<String>(list));
    test("Wynik Arrays.asList()", list);
    test("Wynik unmodifiableList()",
      Collections.unmodifiableList(
        new ArrayList<String>(list)));
  }
} /* Output:
--- Modyfikowalna kopia ---
--- Wynik Arrays.asList() ---
retainAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
--- Wynik unmodifiableList() ---
retainAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
List.set(): java.lang.UnsupportedOperationException
*///:~
```

Ponieważ metoda `Arrays.asList()` zwraca listę realizowaną jako tablica o ustalonym rozmiarze, więc operacjami obsługiwanymi są tylko te, które nie zmieniają rozmiaru tablicy. Z drugiej strony, gdyby do wyrażenia tego odmiennego rodzaju zachowania ustanowić osobny interfejs (nazwany na przykład `FixedSizeList`), stanowiłoby to przyczynek do wzrostu stopnia skomplikowania struktury interfejsów i wkrótce nie byłoby wiadomo, jak korzystać z biblioteki.

Należy zauważyć, że zawsze można przekazać wyniki zwrócone przez metodę `Arrays.asList()` jako argument wywołania konstruktora dowolnego podtypu `Collection` (albo użyć metody `addAll()`, ewentualnie statycznej metody `Collections.addAll()`) w celu utworzenia zwyczajnego kontenera, umożliwiającego wykorzystanie wszystkich dostępnych metod. Takie wywołanie utworzy nową strukturę danych, której rozmiar będzie już mógł się zmieniać.

„Niemodyfikowalne" metody klasy `Collections` ujmują kontener w osłonę, która zgłosi wyjątek `UnsupportedOperationException` za każdym razem, kiedy na rzecz wynikowego kontenera wywołana zostanie dowolna metoda, która w jakikolwiek sposób modyfikuje

kontener. Zadaniem tych metod jest więc tworzenie „niemodyfikowalnych" obiektów kontenerów. Pełna lista takich metod klasy `Collections` zostanie zaprezentowana później.

Ostatni blok `try` w metodzie `test()` bada metodę `set()` wchodzącą w skład interfejsu `List`. To o tyle ciekawe, że ilustruje poręczność wynikającą z selektywności techniki „nieobsługiwanych operacji" — wynikowy „interfejs" obiektów zwracanych przez metody `Arrays.asList()` i `Collections.unmodifiableList()` może się różnić jedną metodą. Wywołanie `Arrays.asList()` zwraca listę o ustalonym rozmiarze, podczas gdy metoda `Collections.unmodifiableList()` zwraca listę, której nie można w ogóle zmieniać. Na wyjściu programu widać, że lista wynikowa metody `Arrays.asList()` dopuszcza *modyfikowanie* elementów, byle nie zmieniać rozmiaru samej listy. Z kolei wynik wywołania `unmodifiableList()` nie powinien być modyfikowany w żaden sposób. Gdyby zaangażować do tego interfejsy, potrzebowalibyśmy dwóch osobnych interfejsów, różniących się jedynie obecnością zdatnej do wywołania metody `set()`. A dla potrzeb kolejnych rozmaitych niemodyfikowalnych podtypów `Collection` trzeba by było powoływać kolejne, osobne interfejsy.

Dokumentacja metody przyjmującej argument w postaci kontenera powinna wyszczególniać, które z opcjonalnych metod muszą być zaimplementowane, a które są opcjonalne.

Ćwiczenie 6. Zauważ, że `List` dysponuje dodatkowymi operacjami „opcjonalnymi" niedostępnymi w interfejsie `Collection`. Napisz wersję programu *Unsupported.java*, która przetestuje te dodatkowe operacje opcjonalne kontenerów `List` (2).

Interfejs List

Wiesz już, że interfejs `List` jest raczej prosty w użyciu: w większości przypadków jego obsługa sprowadza się do wywoływania metody `add()` do wstawiania obiektów, `get()` do ich pobierania pojedynczo oraz metody `iterator()` — by uzyskać `Iterator` do sekwencji elementów.

Metody wykorzystywane w poniższym przykładzie dotyczą różnych aspektów obsługi list: od operacji obsługiwanych przez wszystkie implementacje `List` (metoda `basicTest()`), w tym stosowanie iteratorów (metoda `iterMotion()`), po operacje modyfikacji elementów za pośrednictwem iteratora (`iterManipulation()`), podglądanie efektów modyfikacji listy (`testVisual()`) i operacje charakterystyczne dla list `LinkedList`:

```
//: containers/Lists.java
// Co można zrobić z kontenerami List.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Lists {
  private static boolean b;
  private static String s;
  private static int i;
  private static Iterator<String> it;
  private static ListIterator<String> lit;
  public static void basicTest(List<String> a) {
```

```
    a.add(1, "x"); // Wstawienie na pozycję 1
    a.add("x"); // Dodanie na koniec
    // Dodanie kolekcji:
    a.addAll(Countries.names(25));
    // Dodanie kolekcji od pozycji 3:
    a.addAll(3, Countries.names(25));
    b = a.contains("1"); // Czy mamy tu taki element?
    // Czy mamy tu taką kolekcję?
    b = a.containsAll(Countries.names(25));
    // Listy obsługują dostęp swobodny -- mało kosztowny dla
    // podtypu ArrayList, kosztowny dla LinkedList:
    s = a.get(1); // Pobranie (typowanego) obiektu z pozycji 1
    i = a.indexOf("1"); // Pobranie indeksu obiektu
    b = a.isEmpty(); // Czy lista zawiera cokolwiek?
    it = a.iterator(); // Zwyczajny iterator
    lit = a.listIterator(); // ListIterator
    lit = a.listIterator(3); // Zaczyna od pozycji 3
    i = a.lastIndexOf("1"); // Ostatnie dopasowanie
    a.remove(1); // Usunięcie pozycji 1
    a.remove("3"); // Usunięcie danego obiektu
    a.set(1, "y"); // Ustawienie pozycji 1 na "y"
    // Zachowanie elementów wymienionych w ostatnim argumencie
    // (część wspólna dwóch zbiorów):
    a.retainAll(Countries.names(25));
    // Usunięcie wszystkiego, co znajduje się w argumencie:
    a.removeAll(Countries.names(25));
    i = a.size(); // Jaki jest rozmiar listy?
    a.clear(); // Usunięcie wszystkich elementów
  }
  public static void iterMotion(List<String> a) {
    ListIterator<String> it = a       erator();
    b = it.hasNext();
    b = it.hasPrevious();
    s = it.next();
    i = it.nextIndex();
    s = it.previous();
    i = it.previousIndex();
  }
  public static void iterManipulation(List<String> a) {
    ListIterator<String> it = a.listIterator();
    it.add("47");
    // Po add() trzeba przesunąć iterator na następną pozycję:
    it.next();
    // Usunięcie elementu zza elementu ostatnio dodanego:
    it.remove();
    // Po remove() trzeba przesunąć iterator na następną pozycję:
    it.next();
    // Zmiana elementu za elementem usuniętym:
    it.set("47");
  }
  public static void testVisual(List<String> a) {
    print(a);
    List<String> b = Countries.names(25);
    print("b = " + b);
    a.addAll(b);
    a.addAll(b);
    print(a);
    // Wstawianie, usuwanie i zastępowanie elementów
```

```
    // za pomocą iteratora ListIterator:
    ListIterator<String> x = a.listIterator(a.size()/2);
    x.add("jeden");
    print(a);
    print(x.next());
    x.remove();
    print(x.next());
    x.set("47");
    print(a);
    // Przeglądanie listy wspak:
    x = a.listIterator(a.size());
    while(x.hasPrevious())
      printnb(x.previous() + " ");
    print();
    print("koniec testVisual");
  }
  // Niektóre operacjerzeczy potrafi jedynie LinkedList:
  public static void testLinkedList() {
    LinkedList<String> ll = new LinkedList<String>();
    ll.addAll(Countries.names(25));
    print(ll);
    // Jak stos: odkładamy na stos:
    ll.addFirst("jeden");
    ll.addFirst("dwa");
    print(ll);
    // "Podglądanie" szczytu stosu:
    print(ll.getFirst());
    // Zdejmowanie elementu ze stosu:
    print(ll.removeFirst());
    print(ll.removeFirst());
    // Jak kolejka: wyciąganie elementów z końca kolejki:
    print(ll.removeLast());
    print(ll);
  }
  public static void main(String[] args) {
    // Za każdym razem tworzy i wypełnia nową listę:
    basicTest(
      new LinkedList<String>(Countries.names(25)));
    basicTest(
      new ArrayList<String>(Countries.names(25)));
    iterMotion(
      new LinkedList<String>(Countries.names(25)));
    iterMotion(
      new ArrayList<String>(Countries.names(25)));
    iterManipulation(
      new LinkedList<String>(Countries.names(25)));
    iterManipulation(
      new ArrayList<String>(Countries.names(25)));
    testVisual(
      new LinkedList<String>(Countries.names(25)));
    testLinkedList();
  }
} /* (Execute to see output) *///:~
```

W metodach `basicTest()` i `iterMotion()` wywołania mają demonstrować właściwą składnię, więc choć ich wartości zwracane są przechwytywane, nie są nijak wykorzystywane. W niektórych przypadkach wartości zwracane nie są w ogóle przechwytywane. Szczegółowych informacji o stosowaniu każdej z tych metod należy poszukać w dokumentacji JDK.

Ćwiczenie 7. Utwórz kontenery `ArrayList` i `LinkedList` i wypełnij je za pomocą generatora `Countries.names()`. Wypisz zawartość obu list za pomocą zwyczajnego iteratora (`Iterator`), następnie wstaw jedną listę do drugiej za pomocą iteratora `ListIterator`. Potem wykonaj wstawianie, zaczynając od końca listy i idąc wstecz (4).

Ćwiczenie 8. Utwórz uogólnioną klasę listy jednokierunkowej o nazwie `SList`, która (dla uproszczenia) *nie będzie* implementować interfejsu `List`. Każdy element listy w postaci obiektu `Link` powinien zawierać referencję następnego elementu, ale nie posiadać dostępu do elementu poprzedniego (dla porównania, `LinkedList` to lista dwukierunkowa, co oznacza, że każdy element jest połączony z następnym i poprzednim). Utwórz własny iterator `SListIterator`, który (znów dla uproszczenia) nie będzie implementował interfejsu `ListIterator`. Poza metodą `toString()` jedyną metodą klasy `SList` powinna być metoda `iterator()`, zwracająca `SListIterator`. Dodawanie i usuwanie elementów listy `SList` powinno odbywać się wyłącznie za pośrednictwem iteratora `SListIterator`. Napisz program pokazujący działanie kontenera i jego iteratora (7).

Kontenery Set a kolejność elementów

Przykłady wykorzystujące kontener Set z rozdziału „Kolekcje obiektów" stanowiły dobre wprowadzenie do operacji dających się wykonać na kontenerach Set. W przykładach tych wykorzystywałem jednak wyłącznie predefiniowane typy Javy, jak `Integer` czy `String`, dobrze przystosowane do stosowania w roli typów elementów kontenerów. Tymczasem, jeśli chcemy umieszczać w kontenerach obiekty własnych typów, musimy pamiętać, że kontener Set musi zarządzać ich kolejnością. Postać tego zarządzania różni się pomiędzy implementacjami interfejsu Set. Różne implementacje nie tylko przejawiają rozmaite zachowania, ale też narzucają różne wymagania odnośnie typu obiektów zdatnych do przechowywania w poszczególnych odmianach kontenerów Set:

Implementacja	Zachowanie, wymagania
`Set` (interfejs)	Każdy z elementów dodawanych do zbioru musi być unikatowy, dodanie duplikatu nie powiedzie się. Elementy umieszczane w `Set` muszą definiować metodę `equals()` w celu ustalenia ich unikatowości. Zbiór `Set` ma dokładnie te same metody co `Collection`. Interfejs `Set` nie zapewnia utrzymywania swoich elementów w żadnym szczególnym porządku.
`HashSet*`	Dla zbiorów, dla których istotny jest krótki czas lokalizacji elementu. Elementy przechowywane muszą również definiować metodę `hashCode()`.
`TreeSet`	Zbiór uporządkowany na podstawie drzewa. Dzięki niemu można pobierać uporządkowany ciąg elementów.
`LinkedHashSet`	Cechuje się taką samą szybkością lokalizacji elementów co klasa `HashSet`, jednak zachowuje przy tym oryginalną kolejność dodawania elementów. Działanie klasy opiera się na wykorzystaniu listy połączonej. Dlatego przy przeglądaniu zawartości obiektów `Set` wyniki są pobierane w kolejności, w jakiej były dodawane do kontenera. Elementy przechowywane muszą definiować metodę `hashCode()`.

Gwiazdka przy `HashSet` sygnalizuje, że przy braku ewentualnych dodatkowych ograniczeń należałoby tę implementację stosować jako preferowaną.

Procesowi definiowania metody `hashCode()` przyjrzymy się w dalszej części rozdziału. Niezależnie od implementacji obiekty przechowywane w kontenerach Set muszą implementować metodę `equals()`, tymczasem metoda `hashCode()` potrzebna jest tylko tym klasom, które mają być przechowywane w implementacjach `HashSet` (co jest dość częste, skoro ma to być implementacja preferowana) i `LinkedHashSet`; jednak kanon praktyk programistycznych wymaga, aby przesłanianiu metody `equals()` towarzyszyło również przesłanianie metody `hashCode()`.

Poniższy przykład demonstruje metody, które trzeba zdefiniować, aby móc skutecznie umieszczać obiekty danego typu w kontenerach implementujących interfejs Set:

```
// : containers/TypesForSets.java
// Metody niezbędne w typach obiektów umieszczanych w kontenerach Set.
import java.util.*;

class SetType {
  int i;
  public SetType(int n) { i = n; }
  public boolean equals(Object o) {
    return o instanceof SetType && (i == ((SetType)o).i);
  }
  public String toString() { return Integer.toString(i); }
}

class HashType extends SetType {
  public HashType(int n) { super(n); }
  public int hashCode() { return i; }
}

class TreeType extends SetType
implements Comparable<TreeType> {
  public TreeType(int n) { super(n); }
  public int compareTo(TreeType arg) {
    return (arg.i < i ? -1 : (arg.i == i ? 0 : 1));
  }
}

public class TypesForSets {
  static <T> Set<T> fill(Set<T> set, Class<T> type) {
    try {
      for(int i = 0; i < 10; i++)
          set.add(
            type.getConstructor(int.class).newInstance(i));
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
    return set;
  }
  static <T> void test(Set<T> set, Class<T> type) {
    fill(set, type);
    fill(set, type); // Próba dodania duplikatów
    fill(set, type);
    System.out.println(set);
  }
  public static void main(String[] args) {
    test(new HashSet<HashType>(), HashType.class);
```

```
    test(new LinkedHashSet<HashType>(), HashType.class);
    test(new TreeSet<TreeType>(), TreeType.class);
    // To nie zadziała:
    test(new HashSet<SetType>(), SetType.class);
    test(new HashSet<TreeType>(), TreeType.class);
    test(new LinkedHashSet<SetType>(), SetType.class);
    test(new LinkedHashSet<TreeType>(), TreeType.class);
    try {
      test(new TreeSet<SetType>(), SetType.class);
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
    try {
      test(new TreeSet<HashType>(), HashType.class);
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
  }
} /* Output: (Sample)
[2, 4, 9, 8, 6, 1, 3, 7, 5, 0]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[1, 4, 7, 5, 6, 8, 2, 0, 3, 4, 4, 2, 1, 0, 2, 9, 8, 6, 5, 9, 6, 3, 1, 7, 7, 8, 5, 0, 3, 9]
[9, 2, 2, 4, 4, 1, 8, 9, 0, 8, 0, 6, 6, 7, 7, 7, 6, 5, 9, 5, 2, 3, 1, 1, 3, 5, 8, 3, 4, 0]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
java.lang.ClassCastException: SetType cannot be cast to java.lang.Comparable
java.lang.ClassCastException: HashType cannot be cast to java.lang.Comparable
*///:~
```

Aby sprawdzić, które metody są niezbędne w typach elementów poszczególnych implementacji interfejsu Set, a przy okazji uniknąć powielania kodu, tworzymy trzy klasy. Klasa bazowa SetType przechowuje wartość typu int, implementując też stosowną metodę toString(). Skoro wszystkie typy przechowywane w kontenerach Set muszą posiadać metodę equals(), również ta metoda trafiła do klasy bazowej. Porównanie obiektów klas odbywa się na podstawie wartości pola int i.

Klasa HashType dziedziczy po SetType i uzupełnia typ o definicję metody hashCode(), niezbędną w przypadku obiektów umieszczanych w implementacjach interfejsu Set z haszowaniem.

Interfejs Comparable, implementowany w podklasie TreeType, jest niezbędny dla obiektów mających znajdować zastosowanie w kontenerach sortowanych, jak SortedSet (którego jedyną biblioteczną implementacją jest TreeSet). Zauważ, że w metodzie compareTo() nie użyłem bynajmniej oczywistej i narzucającej się formy return i-i2. To częsty błąd programistyczny; taka implementacja zadziałałaby jedynie wtedy, gdyby i i i2 były wartościami typu int *bez znaku* (choć Java i tak *nie* udostępnia słowa unsigned). W przypadku typu int ze znakiem, właściwego dla Javy, taka implementacja będzie nieskuteczna, bo typ int nie jest na tyle pojemny, aby reprezentować różnicę dwóch wartości typu int. Kiedy i będzie największą możliwą wartością dodatnią, a j najmniejszą możliwą wartością ujemną, różnica i-j przepełni typ int i przyjmie postać wartości ujemnej, co zafałszuje porównanie.

Od metody compareTo() oczekuje się zazwyczaj porządkowania zgodnego z naturalnym porządkiem wartości danego typu, spójnym z porządkiem ustalanym za pomocą metody equals(). Jeśli metoda equals() zwraca dla jakiegoś porównania true, to dla tego samego porównania metoda compareTo() powinna zwrócić wartość zero; z kolei dla porównania, w którym equals() zwraca false, compareTo() powinna dawać wartość niezerową.

W klasie TypesForSets metody test() i fill() zostały zdefiniowane z użyciem uogólnień, co miało zapobiec powielaniu kodu. W celu zweryfikowania zachowania kontenera Set, metoda test() trzykrotnie wywołuje na rzecz testowanego zbioru metodę fill() z identycznym argumentem, próbując wstawić do zbioru duplikaty elementów. Sama metoda fill() przyjmuje kontener Set obiektów dowolnego typu oraz obiekt Class tegoż typu. Ten ostatni służy do wykrycia konstruktora przyjmującego argument typu int i wywołania tego konstruktora w celu wypełnienia kontenera obiektami danego typu.

Na wyjściu widać, że kontener HashSet przechowuje elementy w jakiejś tajemniczej, nieokreślonej kolejności (sposób jej ustalania wyjaśni się w dalszej części rozdziału), kontener LinkedHashSet przechowuje elementy w kolejności zgodnej z kolejnością wstawiania, a TreeSet przechowuje je w kolejności odpowiadającej ich wartościom (z racji sposobu implementowania metody compareTo() jest to porządek malejących wartości).

Próba użycia typów, które nie obsługują w sposób właściwy operacji wymaganych przez kontenery Set, to bardzo zły pomysł. Umieszczenie obiektów typu SetType albo TreeType, pozbawionych metody hashCode(), w którymkolwiek z kontenerów z haszowaniem doprowadzi do dublowania elementów, co jest sprzeczne z podstawową cechą kontenerów Set (jakby nie było — zbiorów). To bardzo przykry błąd, bo nie objawia się nawet jako wyjątek czasu wykonania. Jednak takie zachowanie, choć niepoprawne, jest możliwe i dozwolone. Jedynym niezawodnym sposobem zapewnienia poprawności takiego programu jest włączenie do systemu kompilacji wyczerpujących testów jednostkowych (zobacz suplement publikowany pod adresem *http://MindView.net/Books/BetterJava*).

Próba zastosowania w kontenerze TreeSet typu bez implementacji interfejsu Comparable daje znacznie wyraźniejszy efekt: próba użycia elementów kontenera TreeSet jako obiektów Comparable spowoduje wyjątek.

SortedSet

Zbiór SortedSet (którego jedyną dostępną implementacją jest TreeSet), gwarantuje uporządkowanie przechowywanych elementów, dzięki czemu zyskujemy dodatkowe funkcje dostarczane poprzez metody interfejsu SortedSet:

Metoda	Działanie
Comparator comparator()	Zwraca interfejs porównujący Comparator dla danego zbioru Set lub null w przypadku porządku naturalnego.
Object first()	Podaje najmniejszy element.
Object last()	Podaje największy element.
SortedSet subSet (odElementu, doElementu)	Zwraca fragment zbioru obejmujący elementy od wartości odElementu włącznie do doElementu (bez tego ostatniego).

Metoda	Działanie
`SortedSet headSet(doElementu)`	Zwraca fragment zbioru o wartościach mniejszych niż wartość elementu `doElementu`.
`SortedSet tailSet(odElementu)`	Zwraca fragment danego zbioru obejmujący elementy większe lub równe wartości `odElementu`.

Oto prosty przykład:

```
//: containers/SortedSetDemo.java
// Co można zrobić z kontenerem TreeSet.
import java.util.*;
import static net.mindview.util.Print.*;

public class SortedSetDemo {
  public static void main(String[] args) {
    SortedSet<String> sortedSet = new TreeSet<String>();
    Collections.addAll(sortedSet,
      "jeden dwa trzy cztery pięć sześć siedem osiem"
        .split(" "));
    print(sortedSet);
    String low = sortedSet.first();
    String high = sortedSet.last();
    print(low);
    print(high);
    Iterator<String> it = sortedSet.iterator();
    for(int i = 0; i <= 6; i++) {
      if(i == 3) low = it.next();
      if(i == 6) high = it.next();
      else it.next();
    }
    print(low);
    print(high);
    print(sortedSet.subSet(low, high));
    print(sortedSet.headSet(high));
    print(sortedSet.tailSet(low));
  }
} /* Output:
[cztery, dwa, jeden, osiem, pięć, siedem, sześć, trzy]
cztery
trzy
osiem
trzy
[osiem, pięć, siedem, sześć]
[cztery, dwa, jeden, osiem, pięć, siedem, sześć]
[osiem, pięć, siedem, sześć, trzy]
*///:~
```

Zauważ, że `SortedSet` oznacza: „posortowany zgodnie z funkcją porównującą obiektu", a nie „uporządkowany w kolejności dodawania". Zachowanie porządku wstawiania zapewnia kontener `LinkedHashSet`.

Ćwiczenie 9. Zastosuj generator `RandomGenerator.String` do wypełnienia kontenera `TreeSet`, ale tak, aby wymusić kolejność alfabetyczną. Sprawdź poprawność porządkowania, wypisując zawartość kontenera (2).

Ćwiczenie 10. Bazując na implementacji `LinkedList`, zdefiniuj własną wersję `SortedSet` (7).

Kolejki

Poza klasami mającymi zastosowanie w aplikacjach wielowątkowych jedynymi kolejkami (implementacjami interfejsu Queue) w Javie SE5 są `LinkedList` i `PriorityQueue`, różniące się zachowaniem odnośnie porządkowania elementów (różnice w wydajności są mniej istotne). Oto prosty przykład angażujący większość implementacji kolejek (nie wszystkie nadają się do zastosowania w takim przykładzie) wraz z tymi przeznaczonymi do pracy współbieżnej. Elementy wstawia się do jednego końca kolejki, a wyjmuje się je z drugiego końca:

```
//: containers/QueueBehavior.java
// Porównanie zachowania wybranych kolejek
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

public class QueueBehavior {
  private static int count = 10;
  static <T> void test(Queue<T> queue, Generator<T> gen) {
    for(int i = 0; i < count; i++)
      queue.offer(gen.next());
    while(queue.peek() != null)
      System.out.print(queue.remove() + " ");
    System.out.println();
  }
  static class Gen implements Generator<String> {
    String[] s = ("jeden dwa trzy cztery pięć sześć siedem " +
      "osiem dziewięć dziesięć").split(" ");
    int i;
    public String next() { return s[i++]; }
  }
  public static void main(String[] args) {
    test(new LinkedList<String>(), new Gen());
    test(new PriorityQueue<String>(), new Gen());
    test(new ArrayBlockingQueue<String>(count), new Gen());
    test(new ConcurrentLinkedQueue<String>(), new Gen());
    test(new LinkedBlockingQueue<String>(), new Gen());
    test(new PriorityBlockingQueue<String>(), new Gen());
  }
} /* Output:
jeden dwa trzy cztery pięć sześć siedem osiem dziewięć dziesięć
cztery dwa dziesięć dziewięć jeden osiem pięć siedem sześć trzy
jeden dwa trzy cztery pięć sześć siedem osiem dziewięć dziesięć
jeden dwa trzy cztery pięć sześć siedem osiem dziewięć dziesięć
jeden dwa trzy cztery pięć sześć siedem osiem dziewięć dziesięć
cztery dwa dziesięć dziewięć jeden osiem pięć siedem sześć trzy
*///:~
```

Widać, że poza kolejkami priorytetowymi kolejki udostępniają przechowywane elementy w tej samej kolejności, w jakiej je przyjmują.

Kolejki priorytetowe

Kolejki priorytetowe były już prezentowane w rozdziale „Kolekcje obiektów". Tutaj omówimy je na przykładzie listy zadań do zrobienia, na której każdy obiekt będzie zawierał ciąg znaków i wartość priorytetu rozróżniającą zadania najważniejsze od drugorzędnych. Właściwa kolejność obiektów w kolejce jest kwestią odpowiedniej implementacji interfejsu `Comparable` dla typu elementów:

```
//: containers/ToDoList.java
// Bardziej zaawansowane zastosowanie kolejki PriorityQueue.
import java.util.*;

class ToDoList extends PriorityQueue<ToDoList.ToDoItem> {
  static class ToDoItem implements Comparable<ToDoItem> {
    private char primary;
    private int secondary;
    private String item;
    public ToDoItem(String td, char pri, int sec) {
      primary = pri;
      secondary = sec;
      item = td;
    }
    public int compareTo(ToDoItem arg) {
      if(primary > arg.primary)
        return +1;
      if(primary == arg.primary)
        if(secondary > arg.secondary)
          return +1;
        else if(secondary == arg.secondary)
          return 0;
      return -1;
    }
    public String toString() {
      return Character.toString(primary) +
        secondary + ": " + item;
    }
  }
  public void add(String td, char pri, int sec) {
    super.add(new ToDoItem(td, pri, sec));
  }
  public static void main(String[] args) {
    ToDoList toDoList = new ToDoList();
    toDoList.add("Wynieść śmieci", 'C', 4);
    toDoList.add("Nakarmić psa", 'A', 2);
    toDoList.add("Nakarmić papugę", 'B', 7);
    toDoList.add("Skosić trawnik", 'C', 3);
    toDoList.add("Podlać trawnik", 'A', 1);
    toDoList.add("Nakarmić kota", 'B', 1);
    while(!toDoList.isEmpty())
      System.out.println(toDoList.remove());
  }
} /* Output:
A1: Podlać trawnik
A2: Nakarmić psa
B1: Nakarmić kota
B7: Nakarmić papugę
C3: Skosić trawnik
C4: Wynieść śmieci
*///:~
```

Jak widać, elementy wstawiane do kolejki zostały automatycznie uporządkowane według poziomu ważności.

Ćwiczenie 11. Utwórz klasę zawierającą pole typu `Integer` inicjalizowane wartością z zakresu od 0 do 100, otrzymywaną z generatora `java.util.Random`. Zaimplementuj dla klasy interfejs `Comparable`, porównujący egzemplarze klasy na bazie wartości pola typu `Integer`. Wypełnij obiektami tej klasy kolejkę priorytetową `PriorityQueue` i za pomocą metody `poll()` pokaż, że kolejka uporządkowała obiekty według wartości tego pola (2).

Kolejki dwukierunkowe

Kolejka dwukierunkowa (ang. *deque*) to taka, do której można wstawiać elementy i je z niej wyjmować z obu końców. Operacje charakterystyczne dla kolejki dwukierunkowej są implementowane w kontenerze `LinkedList`, jednak w standardowej bibliotece kontenerów Javy nie istnieje coś takiego jak jawny interfejs kolejek dwukierunkowych. Siłą rzeczy kontener `LinkedList` nie może implementować żadnego takiego interfejsu, a więc nie można rzutować obiektu kontenera w górę na typ `Deque`, jak to było możliwe w poprzednim przykładzie w przypadku kolejki zwykłej — `Queue`. Można natomiast utworzyć własną klasę `Deque` na bazie kompozycji i najzwyczajniej wyeksponować w niej stosowne metody kontenera `LinkedList`:

```
// : net/mindview/util/Deque.java
// Tworzenie kolejki dwukierunkowej (Creating)
// na bazie klasy LinkedList.
package net.mindview.util;
import java.util.*;

public class Deque<T> {
  private LinkedList<T> deque = new LinkedList<T>();
  public void addFirst(T e) { deque.addFirst(e); }
  public void addLast(T e) { deque.addLast(e); }
  public T getFirst() { return deque.getFirst(); }
  public T getLast() { return deque.getLast(); }
  public T removeFirst() { return deque.removeFirst(); }
  public T removeLast() { return deque.removeLast(); }
  public int size() { return deque.size(); }
  public String toString() { return deque.toString(); }
  // I inne metody, w miarę potrzeb...
} ///:~
```

Podejmując próby wykorzystania takiej klasy w swoich programach, zorientujesz się zapewne, że aby klasa była praktyczna, trzeba ją uzupełnić o szereg innych metod.

Tymczasem możemy przetestować nową klasę:

```
// : containers/DequeTest.java
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DequeTest {
  static void fillTest(Deque<Integer> deque) {
    for(int i = 20; i < 27; i++)
      deque.addFirst(i);
    for(int i = 50; i < 55; i++)
      deque.addLast(i);
```

```
  }
  public static void main(String[] args) {
    Deque<Integer> di = new Deque<Integer>();
    fillTest(di);
    print(di);
    while(di.size() != 0)
      printnb(di.removeFirst() + " ");
    print();
    fillTest(di);
    while(di.size() != 0)
      printnb(di.removeLast() + " ");
  }
} /* Output:
[26, 25, 24, 23, 22, 21, 20, 50, 51, 52, 53, 54]
26 25 24 23 22 21 20 50 51 52 53 54
54 53 52 51 50 20 21 22 23 24 25 26
*///:~
```

Potrzeba wstawiania i wyjmowania elementów z obu stron kolejki jest jednak stosunkowo rzadka, więc Deque ma znacznie mniej zastosowań niż zwykła kolejka Queue.

Kontenery asocjacyjne

W rozdziale „Kolekcje obiektów" przedstawione zostały podstawowe cechy *kontenerów asocjacyjnych* (Map). Wiadomo już, że chodzi o kontenery kojarzące klucze z wartościami (a więc przechowujące pary) i pozwalające na wyszukiwanie wartości na podstawie kluczy. Standardowa biblioteka Javy zawiera szereg implementacji interfejsu Map: HashMap, TreeMap, LinkedHashMap, WeakHashMap, ConcurentHashMap i IdentityHashMap. Wszystkie udostępniają podstawowy interfejs kontenerów asocjacyjnych, różnią się jednak zachowaniem, w tym efektywnością, kolejnością przechowywania i prezentowania elementów (par), czasem przetrzymywania obiektów w kontenerze, sprawnością i bezpieczeństwem w programach współbieżnych i stwierdzaniem równości kluczy. O roli kontenerów asocjacyjnych w rozwiązywaniu zadań programistycznych świadczy choćby sama liczba dostępnych, standardowych implementacji.

Warto więc poznać kontenery asocjacyjne bliżej, przyglądając się sposobowi konstruowania tablicy asocjacyjnej. Oto prosta przykładowa implementacja takiej tablicy:

```
//: containers/AssociativeArray.java
// Kojarzenie kluczy z wartościami.
import static net.mindview.util.Print.*;

public class AssociativeArray<K,V> {
  private Object[][] pairs;
  private int index;
  public AssociativeArray(int length) {
    pairs = new Object[length][2];
  }
  public void put(K key, V value) {
    if(index >= pairs.length)
      throw new ArrayIndexOutOfBoundsException();
    pairs[index++] = new Object[]{ key, value };
```

```
  }
  @SuppressWarnings("unchecked")
  public V get(K key) {
    for(int i = 0; i < index; i++)
      if(key.equals(pairs[i][0]))
        return (V)pairs[i][1];
    return null; // Brak szukanego klucza
  }
  public String toString() {
    StringBuilder result = new StringBuilder();
    for(int i = 0; i < index; i++) {
      result.append(pairs[i][0].toString());
      result.append(" : ");
      result.append(pairs[i][1].toString());
      if(i < index - 1)
        result.append("\n");
    }
    return result.toString();
  }
  public static void main(String[] args) {
    AssociativeArray<String,String> map =
      new AssociativeArray<String,String>(6);
    map.put("niebo", "niebieskie");
    map.put("trawa", "zielona");
    map.put("ocean", "falujący");
    map.put("drzewo", "wysokie");
    map.put("ziemia", "brązowa");
    map.put("słońce", "gorące");
    try {
      map.put("ekstra", "obiekt"); // Za końcem
    } catch(ArrayIndexOutOfBoundsException e) {
      print("Za dużo obiektów!");
    }
    print(map);
    print(map.get("ocean"));
  }
} /* Output:
Za dużo obiektów!
niebo : niebieskie
trawa : zielona
ocean : falujący
drzewo : wysokie
ziemia : brązowa
słońce : gorące
falujący
*///:~
```

Najważniejszymi metodami kontenerów asocjacyjnych są `put()` i `get()`, ale dla prostoty wypisywania zawartości tablicy przesłoniliśmy również metodę `toString()` tak, aby wypisywała pary klucz-wartość. W ramach testu metoda `main()` wypełnia egzemplarz `AssociativeArray` parami ciągów i wypisuje zawartość wypełnionego kontenera na wyjściu programu; przedtem wyciąga jedną z wartości za pomocą metody `get()`.

Zastosowanie metody `get()` polega na przekazaniu w wywołaniu klucza identyfikującego pożądaną parę. Metoda zwraca wartość skojarzoną z kluczem albo wartość pustą (`null`), jeśli kontener nie zawiera wskazanego klucza. Nasza metoda `get()` stosuje chyba najmniej efektywne podejście do zadania lokalizowania wartości w tablicy: przegląda

zawartość tablicy, porównując klucze poszczególnych par z kluczem zadanym w wywołaniu za pośrednictwem metody `equals()`. Ale nam chodziło o prostotę, a nie wydajność.

Powyższa implementacja jest pouczająca, ale nieefektywna i nieelastyczna, choćby ze względu na stały rozmiar tablicy. Na szczęście kontenery asocjacyjne z biblioteki `java.util` są wolne od podobnych wad.

Ćwiczenie 12. W metodzie `main()` programu *AssociativeArray.java* podmień tablicę asocjacyjną `AssociativeArray` na kontenery `HashMap`, `TreeMap` i `LinkedHashMap` (1).

Ćwiczenie 13. Wykorzystaj program *AssociativeArray.java* do tworzenia licznika wystąpień słów odwzorowującego ciągi znaków (`String`) na liczby całkowite (`Integer`). Za pomocą klasy `net.mindview.util.TextFile` otwórz plik tekstowy i wyodrębnij z niego słowa (na bazie odstępów i znaków przestankowych) i zlicz wystąpienia poszczególnych słów w pliku (4).

Wydajność

Bardzo ważną cechą kontenerów asocjacyjnych, zwanych też odwzorowaniami, jest efektywność ich działania, a liniowe (sekwencyjne) wyszukiwanie klucza wydaje się być dosyć wolne. Tutaj właśnie przyspieszenie zapewnia `HashMap`. Zamiast powolnego poszukiwania klucza stosuje specjalną wartość, zwaną *kodem haszującym*. Kod haszujący jest sposobem na pobranie pewnych informacji o obiekcie i zamianę w „stosunkowo unikatową" wartość typu `int`. Wszystkie obiekty Javy mogą udostępnić swój kod haszujący, a służy do tego metoda `hashCode()` z głównej klasy bazowej `Object`. `HashMap` pobiera właśnie wartość `hashCode()` obiektu i używa jej w celu szybkiego „upolowania" klucza. W rezultacie zyskujemy ogromny wzrost wydajności[6].

Tabela przedstawiona na następnej stronie wymienia podstawowe implementacje kontenera `Map`. Znak gwiazdki przy kontenerze `HashMap` oznacza zalecenie do wykorzystywania go w roli kontenera domyślnego, a to z racji jego optymalizacji pod kątem szybkości. Pozostałe implementacje uwypuklają inne cechy odwzorowań i nie są tak szybkie jak `HashMap`.

Wiedza na temat działania haszowania, jako operacji determinującej sposób przechowywania elementów w kontenerach asocjacyjnych, przyda się z pewnością w praktyce programistycznej, toteż niebawem przyjrzymy się haszowaniu bliżej.

Wymagania odnośnie kluczy par przechowywanych w kontenerach `Map` są identyczne jak wymagania odnośnie elementów kontenerów `Set`. Te zaś były omawiane przy okazji przykładu *TypesForSet.java*. Wszelkie klucze muszą implementować metodę `equals()`.

---

[6] Jeżeli to przyspieszenie wciąż nie spełnia Twoich oczekiwań względem wydajności, to możesz jeszcze bardziej przyspieszyć wyszukiwanie poprzez zaimplementowanie własnego odwzorowania `Map` dla konkretnego typu, aby uniknąć opóźnień spowodowanych rzutowaniem do i z `Object`. Aby uzyskać jeszcze wyższą efektywność, możesz zerknąć do dzieła *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition* autorstwa Donalda Knutha, żeby zastąpić listy z haszowaniem kubełkowym tablicami, które posiadają dwie dodatkowe właściwości: mają zoptymalizowany zapis na dysk oraz mogą zaoszczędzić dużo czasu, samodzielnie tworząc i odśmiecając pojedyncze elementy.

implementacja	Działanie
HashMap*	Implementacja oparta na tablicy haszującej (należy jej używać zamiast Hashtable). Zapewnia wstawianie i lokalizację par w czasie stałym. Zachowanie może być regulowane dzięki konstruktorowi, który pozwala na ustawienie *pojemności* i *współczynnika wypełnienia* tablicy haszującej.
LinkedHashMap	Implementacja podobna do HashMap, lecz podczas jej przeglądania pary są zwracane w kolejności wstawiania lub zaczynając od najdawniej używanych. Działa trochę wolniej od HashMap; wyjątkiem jest tutaj wstawianie elementów, które, dzięki wykorzystaniu listy połączonej do określania wewnętrznej kolejności elementów, jest szybsze.
TreeMap	Implementacja oparta na drzewach binarnych. Kiedy wypiszemy klucze lub zamieszczone pary, to będą one posortowane (w porządku wyznaczonym przez interfejs Comparable lub interfejs Comparator, omówione dalej). Właśnie otrzymanie uporządkowanego wyniku jest główną właściwością TreeMap. TreeMap jest przy tym jedynym odwzorowaniem posiadającym metodę subMap(), która pozwala uzyskać fragment drzewa.
WeakHashMap	Odwzorowanie operujące na *słabych kluczach*, umożliwiające usunięcie z pamięci obiektów przechowywanych w mapie; zaprojektowane w celu rozwiązywania problemów szczególnego typu. Jeśli poza odwzorowaniem w programie nie ma innych odwołań do przechowywanych w nim obiektów, to obiekty te mogą zostać usunięte przez odśmiecacz pamięci.
ConcurrentHashMap	Implementacja Map przystosowana do użycia w aplikacjach współbieżnych z synchronizacją dostępu. Omawiana w rozdziale „Współbieżność".
IdentityHashMap	Odwzorowanie haszujące określające równość kluczy przy wykorzystaniu operatora == zamiast metody equals(). Wykorzystywane wyłącznie do rozwiązywania szczególnego typu problemów, nie nadaje się do zastosowań ogólnych.

Jeśli klucz ma być wykorzystywany w odwzorowaniu haszowanym, powinien również definiować odpowiednio metodę hashCode(). Klucze przeznaczone do stosowania w TreeMap powinny ponadto implementować interfejs Comparable.

Następny przykład pokazuje operacje dostępne za pośrednictwem interfejsu Map na bazie zdefiniowanej wcześniej klasy testowego zbioru danych CountingMapData:

```
//: containers/Maps.java
// Co da się zrobić z kontenerami Map.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Maps {
  public static void printKeys(Map<Integer,String> map) {
    printnb("Rozmiar = " + map.size() + ", ");
    printnb("Klucze: ");
    print(map.keySet()); // Tworzy zbiór (Set) kluczy odwzorowania
  }
  public static void test(Map<Integer,String> map) {
    print(map.getClass().getSimpleName());
    map.putAll(new CountingMapData(25));
    // Kontenery Map wymagają od kluczy zachowania
```

```
    // właściwego dla elementów kontenerów Set:
    map.putAll(new CountingMapData(25));
    printKeys(map);
    // Tworzenie kolekcji wartości:
    printnb("Wartości: ");
    print(map.values());
    print(map);
    print("map.containsKey(11): " + map.containsKey(11));
    print("map.get(11): " + map.get(11));
    print("map.containsValue(\"F0\"): "
      + map.containsValue("F0"));
    Integer key = map.keySet().iterator().next();
    print("Pierwszy klucz odwzorowania: " + key);
    map.remove(key);
    printKeys(map);
    map.clear();
    print("map.isEmpty(): " + map.isEmpty());
    map.putAll(new CountingMapData(25));
    // Operacje na zbiorze lkkluczy modyfikują odwzorowanie:
    map.keySet().removeAll(map.keySet());
    print("map.isEmpty(): " + map.isEmpty());
  }
  public static void main(String[] args) {
    test(new HashMap<Integer,String>());
    test(new TreeMap<Integer,String>());
    test(new LinkedHashMap<Integer,String>());
    test(new IdentityHashMap<Integer,String>());
    test(new ConcurrentHashMap<Integer,String>());
    test(new WeakHashMap<Integer,String>());
  }
} /* Output:
HashMap
Rozmiar = 25, Klucze: [15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 14, 24, 4, 19, 11, 18, 3, 12, 17, 2, 13, 20, 10, 5, 0]
Wartości: [P0, I0, X0, Q0, H0, W0, J0, V0, G0, B0, O0, Y0, E0, T0, L0, S0, D0, M0, R0, C0, N0, U0, K0,
F0, A0]
{15=P0, 8=I0, 23=X0, 16=Q0, 7=H0, 22=W0, 9=J0, 21=V0, 6=G0, 1=B0, 14=O0, 24=Y0, 4=E0, 19=T0,
11=L0, 18=S0, 3=D0, 12=M0, 17=R0, 2=C0, 13=N0, 20=U0, 10=K0, 5=F0, 0=A0}
map.containsKey(11): true
map.get(11): L0
map.containsValue("F0"): true
Pierwszy klucz odwzorowania: 15
Rozmiar = 24, Klucze: [8, 23, 16, 7, 22, 9, 21, 6, 1, 14, 24, 4, 19, 11, 18, 3, 12, 17, 2, 13, 20, 10, 5, 0]
map.isEmpty(): true
map.isEmpty(): true
...
*///:~
```

Metoda `printKeys()` pokazuje, jak pobrać obiekty typu `Collection` z odwzorowania `Map`. Metoda `keySet()` zwraca zbiór `Set` zawierający wszystkie klucze odwzorowania. Podobnie jest w przypadku `values()`, która zwraca kontener `Collection` zawierający wszystkie wartości z odwzorowania (zauważ, że klucze muszą być unikatowe, podczas gdy wartości mogą się powtarzać). Z uwagi na to, że za tymi kontenerami stoi odwzorowanie `Map`, każda zmiana w kontenerze znajdzie odzwierciedlenie w powiązanym odwzorowaniu.

Reszta programu to proste przykłady operacji interfejsu `Map` i test każdego z rodzajów odwzorowań.

Ćwiczenie 14. Pokaż, jak w powyższym programie można wykorzystać klasę `java.util.Properties` (3).

SortedMap

Jeśli dysponujemy interfejsem SortedMap (którego jedyną dostępną implementacją jest TreeMap), to klucze są zawsze uporządkowane. Pozwala to na uzyskanie dodatkowych możliwości dzięki następującym metodom interfejsu SortedMap:

Metoda	Działanie
`Comparator comparator()`	Zwraca interfejs porównujący dla danego odwzorowania Map lub null dla porządku naturalnego.
`Object firstKey()`	Podaje najmniejszy z kluczy.
`Object lastKey()`	Podaje największy z kluczy.
`SortedMap subMap (odKlucza, doKlucza)`	Zwraca fragment odwzorowania obejmujący klucze od wartości odKlucza włącznie do doKlucza (bez niego samego).
`SortedMap headMap(doKlucza)`	Zwraca fragment odwzorowania z kluczami o wartościach mniejszych niż wartość doKlucza.
`SortedMap tailMap(odKlucza)`	Zwraca fragment danego odwzorowania obejmujący klucze większe bądź równe odKlucza.

Oto przykład podobny do *SortedSetDemo.java*, prezentujący to dodatkowe zachowanie klasy TreeMap:

```
// containers/SortedMapDemo.java
// Co można zrobić z kontenerem TreeMap.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class SortedMapDemo {
  public static void main(String[] args) {
    TreeMap<Integer,String> sortedMap =
      new TreeMap<Integer,String>(new CountingMapData(10));
    print(sortedMap);
    Integer low = sortedMap.firstKey();
    Integer high = sortedMap.lastKey();
    print(low);
    print(high);
    Iterator<Integer> it = sortedMap.keySet().iterator();
    for(int i = 0; i <= 6; i++) {
      if(i == 3) low = it.next();
      if(i == 6) high = it.next();
      else it.next();
    }
    print(low);
    print(high);
    print(sortedMap.subMap(low, high));
    print(sortedMap.headMap(high));
    print(sortedMap.tailMap(low));
  }
} /* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0}
0
9
3
```

```
7
{3=D0, 4=E0, 5=F0, 6=G0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0}
{3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0}
*///:~
```

W tym przykładzie pary są sortowane na podstawie kluczy. Ponieważ właśnie taki jest sens uporządkowania w klasie TreeMap, zasadne jest tu pojęcie „pozycji"; a zatem można wyróżnić element pierwszy, ostatni, jak również „pod-odwzorowania".

LinkedHashMap

W celu zapewnienia szybkości działania klasa LinkedHashMap zawsze wykorzystuje funkcję haszującą, jednak podczas przeglądania zawartości zwraca pary w kolejności, w jakiej były dodawane (metoda System.out.println() kolejno przegląda i wyświetla zawartość odwzorowania, dzięki czemu można zobaczyć rezultaty operacji). Co więcej, klasę można skonfigurować (w konstruktorze) w taki sposób, by do określania kolejności dostępu wykorzystywała algorytm LRU (czyli najdawniej używany, ang. *least-recently-used*). Dzięki temu elementy, które nie były używane (i są kandydatami do usunięcia), były widoczne na samym początku listy. Rozwiązanie to ułatwia tworzenie programów, które cyklicznie, co pewien czas dokonują „sprzątania" w celu zwolnienia niepotrzebnie zajmowanej pamięci. Oto program przedstawiający obie te możliwości:

```
//: containers/LinkedHashMapDemo.java
// Co można zrobić z kontenerem LinkedHashMap.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class LinkedHashMapDemo {
  public static void main(String[] args) {
    LinkedHashMap<Integer,String> linkedMap =
      new LinkedHashMap<Integer,String>(
        new CountingMapData(9));
    print(linkedMap);
    // Według czasu ostatniego użycia:
    linkedMap =
      new LinkedHashMap<Integer,String>(16, 0.75f, true);
    linkedMap.putAll(new CountingMapData(9));
    print(linkedMap);
    for(int i = 0; i < 6; i++) // odwołania:
      linkedMap.get(i);
    print(linkedMap);
    linkedMap.get(0);
    print(linkedMap);
  }
} /* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{6=G0, 7=H0, 8=I0, 0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0}
{6=G0, 7=H0, 8=I0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 0=A0}
*///:~
```

Analizując wyniki wygenerowane przez program, można się przekonać, że pary faktycznie są odwiedzane w kolejności ich dodawania i to nawet w przypadku wykorzystania algorytmu LRU. Jednak po „odwiedzeniu" pierwszych siedmiu elementów, pozostałe trzy zostają przeniesione na początek listy. Następnie, po ponownym „odwiedzeniu" elementu jeden, zostaje on przeniesiony na koniec listy.

Haszowanie i kody haszujące

W przykładach z rozdziału „Kolekcje obiektów" w roli kluczy kontenera `HashMap` występowały predefiniowane klasy Javy. Działały świetnie, ponieważ klasy te zawierały wszystko, co konieczne, aby taką rolę spełniać.

Jednakże w przypadku `HashMap` mamy do czynienia z powszechną pułapką, kiedy tworzymy własne klasy mające pełnić funkcję kluczy. Przykładowo rozważmy system prognozujący pogodę, który kojarzy obiekty `Groundhog`[7] (świstak) z obiektami `Prediction` (prognoza). Wydaje się to proste: tworzymy dwie klasy i stosujemy `Groundhog` jako klucz, a `Prediction` jako wartość:

```
// : containers/Groundhog.java
// Wygląda nieźle, ale nie nadaje się do roli klucza w HashMap.

public class Groundhog {
  protected int number;
  public Groundhog(int n) { number = n; }
  public String toString() {
    return "Groundhog nr " + number;
  }
} ///:~

// : containers/Prediction.java
// Prognozowanie pogody za pomocą świstaków.
import java.util.*;

public class Prediction {
  private static Random rand = new Random(47);
  private boolean shadow = rand.nextDouble() > 0.5;
  public String toString() {
    if(shadow)
      return "Jeszcze sześć tygodni zimy!";
    else
      return "Wczesna wiosna!";
  }
} ///:~

// : containers/SpringDetector.java
// Jaka będzie pogoda?
import java.lang.reflect.*;
```

[7] Według amerykańskich tradycji ludowych świstak wychodzi ze swojego legowiska 2 lutego i po długości jego cienia tego dnia można przepowiedzieć pogodę. Jeśli cień świstaka jest widoczny, to będzie jeszcze sześć tygodni chłodu, natomiast w przeciwnym razie będzie wczesna wiosna — *przyp. tłum.*

```
import java.util.*;
import static net.mindview.util.Print.*;

public class SpringDetector {
  // Używa klasy Groundhog albo klasy pochodnej:
  public static <T extends Groundhog>
  void detectSpring(Class<T> type) throws Exception {
    Constructor<T> ghog = type.getConstructor(int.class);
    Map<Groundhog.Prediction> map =
      new HashMap<Groundhog.Prediction>();
    for(int i = 0; i < 10; i++)
      map.put(ghog.newInstance(i), new Prediction());
    print("odwzorowanie = " + map);
    Groundhog gh = ghog.newInstance(3);
    print("Szukanie prognozy dla " + gh);
    if(map.containsKey(gh))
      print(map.get(gh));
    else
      print("Brak klucza: " + gh);
  }
  public static void main(String[] args) throws Exception {
    detectSpring(Groundhog.class);
  }
} /* Output:
odwzorowanie = {Groundhog nr 2=Wczesna wiosna!, Groundhog nr 5=Wczesna wiosna!, Groundhog
nr 4=Jeszcze sześć miesięcy zimy!, Groundhog nr 7=Wczesna wiosna!, Groundhog nr 1=Jeszcze sześć
miesięcy zimy!, Groundhog nr 8=Jeszcze sześć miesięcy zimy!, Groundhog nr 3=Wczesna wiosna!,
Groundhog nr 0=Jeszcze sześć miesięcy zimy!, Groundhog nr 9=Jeszcze sześć miesięcy zimy!,
Groundhog nr 6=Wczesna wiosna!}
Szukanie prognozy dla Groundhog nr 3
Brak klucza: Groundhog nr 3
*///:~
```

Każdy obiekt `Groundhog` otrzymuje numer identyfikacyjny, toteż można odszukać `Prediction` w `HashMap` poprzez żądanie: „Podaj prognozę dla świstaka o numerze 3". Klasa `Prediction` zawiera zmienną typu `boolean`, inicjowaną metodą `java.util.random()` oraz metodę `toString()`, która interpretuje wynik losowania. Metoda `detectSpring()` wykorzystuje mechanizmy refleksji, aby inicjalizować i używać obiektów klasy `Groundhog` lub klas potomnych. Rozwiązanie to może się przydać, gdy stworzymy nową klasę potomną `Groundhog`, aby rozwiązać problemy przedstawione w tym przykładzie.

W metodzie `main()` `HashMap` zostaje wypełniona obiektami `Groundhog` i towarzyszącymi im prognozami `Prediction`. Wypisując `HashMap`, widzimy, iż rzeczywiście została wypełniona. Następnie obiekt `Groundhog` o identyfikatorze 3 (który, jak widać, musi być w odwzorowaniu) jest stosowany jako klucz, by wyszukać dla niego prognozę.

Wygląda to dosyć prosto, ale nie działa — nie udało się wyszukać klucza dla świstaka o numerze 3. Problem polega na tym, że `Groundhog` jest pochodną klasy głównej `Object`, a do wygenerowania kodu haszującego służy metoda `hashCode()` pochodząca właśnie z klasy `Object`, która domyślnie po prostu używa adresu pamięci swojego obiektu. Tym samym pierwszy egzemplarz `Groundhog(3)` *nie* zyskuje kodu haszującego równego kodowi drugiego egzemplarza `Groundhog(3)`, który próbowaliśmy wyszukać.

Można by przypuszczać, że wystarczy tylko przesłonić metodę `hashCode()`. Jednak nadal nie będzie działać, póki nie przesłoni się metody `equals()` będącej częścią klasy `Object`. Metoda ta jest wykorzystana przez `HashMap`, kiedy próbuje określić, czy klucz jest równy któremuś z kluczy tablicy.

Poprawnie zaimplementowana metoda `equals()` musi spełniać pięć poniższych warunków:

1. Musi być zwrotna: dla każdego `x`, `x.equals(x)` ma zwracać wartość `true`.
2. Musi być symetryczna: dla dowolnych `x` i `y`, `x.equals(y)` ma zwracać wartość `true` wtedy i tylko wtedy, gdy `y.equals(x)` zwraca `true`.
3. Musi być przechodnia: dla dowolnych `x`, `y` oraz `z`, jeśli `x.equals(y)` zwraca `true` oraz `y.equals(z)` zwraca `true`, to także `x.equals(z)` powinna zwracać `true`.
4. Musi być spójna: dla dowolnych `x` i `y` wielokrotne wywołania `x.equals(y)` spójnie zwracają wartość `true` bądź wartość `false`, zakładając że żadne informacje używane przy porównywaniu obiektów nie zostały zmienione.
5. Dla dowolnego `x` różnego od `null` wywołanie `x.equals(null)` powinno zwracać `false`.

Jeszcze raz powtórzę, że domyślna implementacja metody `Object.equals()` po prostu porównuje adresy obiektów, dlatego żadne dwa obiekty utworzone przy użyciu wywołania `Groundhog(3)` nie będą sobie równe. A zatem, aby móc używać własnych klas jako kluczy w odwzorowaniach `HashMap`, należy przesłonić obie metody — zarówno `hashCode()`, jak i `equals()` — jak pokazałem w poniższym przykładzie rozwiązującym problem przewidywania pogody na podstawie obserwacji świstaków.

```
// : containers/Groundhog2.java
// Klasa wykorzystywana w roli klucza w HashMap
// musi przesłaniać metody hashCode() i equals().

public class Groundhog2 extends Groundhog {
  public Groundhog2(int n) { super(n); }
  public int hashCode() { return number; }
  public boolean equals(Object o) {
    return o instanceof Groundhog2 &&
      (number == ((Groundhog2)o).number);
  }
} ///:~

// : containers/SpringDetector2.java
// Działające klucze.

public class SpringDetector2 {
  public static void main(String[] args) throws Exception {
    SpringDetector.detectSpring(Groundhog2.class);
  }
} /* Output:
odwzorowanie = {Groundhog nr 2=Wczesna wiosna!, Groundhog nr 4=Jeszcze sześć miesięcy zimy!,
Groundhog nr 9=Jeszcze sześć miesięcy zimy!, Groundhog nr 8=Jeszcze sześć miesięcy zimy!, Groundhog
nr 6=Wczesna wiosna!, Groundhog nr 1=Jeszcze sześć miesięcy zimy!, Groundhog nr 3=Wczesna wiosna!,
Groundhog nr 7=Wczesna wiosna!, Groundhog nr 5=Wczesna wiosna!, Groundhog nr 0=Jeszcze sześć
miesięcy zimy!}
Szukanie prognozy dla Groundhog nr 3
Wczesna wiosna!
*///:~
```

`Groundhog2.hashCode()` jako identyfikator zwraca numer obiektu. W tym przykładzie programista jest odpowiedzialny za zapewnienie, by żadne dwa obiekty `Groundhog` nie występowały z tą samą wartością identyfikatora. Metoda `hashCode()` nie musi zwracać identyfikatora unikatowego (zrozumiesz to za chwilę, czytając ten rozdział), ale metoda `equals()` musi umieć dokładnie rozpoznać, czy dwa obiekty są równoważne. W powyższym przykładzie działanie metody `equals()` bazuje na porównaniu numerów obiektów, a zatem, jeśli kluczami w obiekcie `HashMap` będą dwa obiekty `Groundhog2` posiadające te same numery, to metoda nie będzie działać poprawnie.

Pomimo że wydaje się, iż metoda `equals()` jedynie sprawdza, czy argument jest egzemplarzem klasy `Groundhog2` (stosując słowo kluczowe `instanceof`, które zostało w pełni objaśnione w rozdziale „Informacje o typach"), to `instanceof` w rzeczywistości po cichu dokonuje drugiego sprawdzenia, by zobaczyć, czy obiekt nie jest `null`, gdyż `instanceof` zwraca wartość fałszu, jeśli pierwszy argument jest odwołaniem pustym. Zakładając, że typ jest właściwy i nie ma wartości `null`, porównanie opiera się na numerach `number`. Jeżeli teraz uruchomimy program, okaże się, że daje poprawne wyniki (wiele klas z bibliotek Javy przesłania metody `hashCode()` i `equals()`wersjami opartymi na ich zawartości).

Podczas tworzenia własnych klas do użytku wraz z `HashSet` trzeba zwrócić uwagę na tę samą kwestię, co w przypadku użycia jako klucza dla `HashMap`.

Zasada działania hashCode()

Powyższy przykład jest jedynie wstępem do właściwego zrozumienia rozwiązania problemu. Pokazuje, że jeśli nie nadpisze się metody `hashCode()` i `equals()` dla kluczy, to struktura haszująca (`HashSet`, `HashMap`, `LinkedHashSet` czy `LinkedHashMap`) nie będzie mogła poprawnie działać. Aby uzyskać *dobre* rozwiązanie tego problemu, trzeba zrozumieć, co się dzieje wewnątrz takiej struktury danych z haszowaniem.

Po pierwsze, rozważmy motywy kryjące się za stosowaniem haszowania: chcemy odnaleźć obiekt, stosując inny obiekt. Ale przecież można to również zrobić, stosując `TreeMap`, albo nawet implementując własny kontener `Map`. Poniższy przykład pokazuje takie rozwiązanie, bazujące na dwóch kontenerach `ArrayList`. W przeciwieństwie do programu *AssociativeArray.java* mamy tu pełną implementację interfejsu `Map`, co widać w metodzie `entrySet()`:

```
// : containers/SlowMap.java
// Kontener Map implementowany za pomocą pary kontenerów ArrayList.
import java.util.*;
import net.mindview.util.*;

public class SlowMap<K,V> extends AbstractMap<K,V> {
  private List<K> keys = new ArrayList<K>();
  private List<V> values = new ArrayList<V>();
  public V put(K key, V value) {
    V oldValue = get(key); // Poprzednia wartość albo null
    if(!keys.contains(key)) {
      keys.add(key);
      values.add(value);
    } else
      values.set(keys.indexOf(key), value);
    return oldValue;
  }
```

```
  public V get(Object key) { // Klucz jest typu Object, nie K
    if(!keys.contains(key))
      return null;
    return values.get(keys.indexOf(key));
  }
  public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> set= new HashSet<Map.Entry<K,V>>();
    Iterator<K> ki = keys.iterator();
    Iterator<V> vi = values.iterator();
    while(ki.hasNext())
      set.add(new MapEntry<K,V>(ki.next(), vi.next()));
    return set;
  }
  public static void main(String[] args) {
    SlowMap<String,String> m= new SlowMap<String,String>();
    m.putAll(Countries.capitals(15));
    System.out.println(m);
    System.out.println(m.get("BULGARIA"));
    System.out.println(m.entrySet());
  }
} /* Output:
{CAMEROON=Yaounde, CHAD=N'djamena, CONGO=Brazzaville, CAPE VERDE=Praia,
ALGERIA=Algiers, COMOROS=Moroni, CENTRAL AFRICAN REPUBLIC=Bangui,
BOTSWANA=Gaberone, BURUNDI=Bujumbura, BENIN=Porto-Novo, BULGARIA=Sofia,
EGYPT=Cairo, ANGOLA=Luanda, BURKINA FASO=Ouagadougou, DJIBOUTI=Dijibouti}
Sofia
[CAMEROON=Yaounde, CHAD=N'djamena, CONGO=Brazzaville, CAPE VERDE=Praia,
ALGERIA=Algiers, COMOROS=Moroni, CENTRAL AFRICAN REPUBLIC=Bangui,
BOTSWANA=Gaberone, BURUNDI=Bujumbura, BENIN=Porto-Novo, BULGARIA=Sofia,
EGYPT=Cairo, ANGOLA=Luanda, BURKINA FASO=Ouagadougou, DJIBOUTI=Dijibouti]
*///:~
```

Metoda `put()` upraszcza zamieszczanie kluczy i wartości w odpowiednich listach klasy `ArrayList`. Wedle wymogów interfejsu `Map` metoda powinna zwracać poprzednią wartość dla klucza albo wartość `null`, jeśli klucza nie było jeszcze w kontenerze.

Podobnie metoda `get()` powinna zwracać wartość `null`, jeśli nie znajdzie klucza w kontenerze `SlowMap`. Jeśli zadany klucz występuje w kontenerze, służy do wyszukania wartości indeksu liczbowego wskazującego pozycję w liście kluczy `keys`, z której to pozycji odczytuje się potem położenie wartości skojarzonej z kluczem na liście wartości `values`. Typ klucza w metodzie `get()` to `Object`, a nie typ parametryzowany `K`, jak można by oczekiwać (i jak to było w programie *AssociativeArray.java*). To skutek późnego „wstrzyknięcia" uogólnień do języka Java — gdyby uogólnienia były rdzennym elementem języka, metoda `get()` mogłaby przyjmować klucz zgodny ze swoim parametrem typowym.

Metoda `Map.entrySet()` powinna zwracać obiekty klasy `Map.Entry`. Niemniej jednak `Map.Entry` to interfejs opisujący strukturę zależną od implementacji, więc definiując własny kontener `Map`, należy zdefiniować również własną implementację `Map.Entry`:

```
//: containers/MapEntry.java
// Prosta implementacja Map.Entry dla przykładowych implementacji Map.
import java.util.*;

public class MapEntry<K,V> implements Map.Entry<K,V> {
  private K key;
  private V value;
```

```
  public MapEntry(K key, V value) {
    this.key = key;
    this.value = value;
  }
  public K getKey() { return key; }
  public V getValue() { return value; }
  public V setValue(V v) {
    V result = value;
    value = v;
    return result;
  }
  public int hashCode() {
    return (key==null ? 0 : key.hashCode()) ^
      (value==null ? 0 : value.hashCode());
  }
  public boolean equals(Object o) {
    if(!(o instanceof MapEntry)) return false;
    MapEntry me = (MapEntry)o;
    return
      (key == null ?
       me.getKey() == null : key.equals(me.getKey())) &&
      (value == null ?
       me.getValue()== null : value.equals(me.getValue()));
  }
  public String toString() { return key + "=" + value; }
} ///:~
```

Zadanie przechowywania i udostępniania kluczy i wartości realizuje tu bardzo prosta klasa o nazwie `MapEntry`. Wykorzystywana jest w metodzie `entrySet()` do tworzenia zbioru par klucz-wartość. Metoda `entrySet()` wykorzystuje do przechowywania par kontener `HashSet`, a klasa `MapEntry` opiera się na metodzie `hashCode()` klucza. Choć to bardzo proste rozwiązanie, przechodzące pomyślnie proste testy wykonywane w programie *SlowMap.java*, nie stanowi implementacji poprawnej, a to z powodu wykonywania kopii kluczy i wartości. Właściwa implementacja `entrySet()` powinna udostępniać *perspektywę* (podgląd) odwzorowania, a nie jego kopię; zwracana perspektywa powinna umożliwiać modyfikowanie oryginalnego odwzorowania (a kopia jest od niego niezależna). Okazję do wyeliminowania tej słabości daje ćwiczenie numer 16.

Metoda `equals()` w klasie `MapEntry` musi porównywać i klucze, i wartości. Znaczenie metody `hashCode()` zostanie omówione wkrótce.

Reprezentacja zawartości odwzorowania `SlowMap` w postaci ciągu znaków jest generowana automatycznie przez metodę `toString()` zdefiniowaną w klasie bazowej `AbstractMap`.

W metodzie `SlowMap.main()` następuje wypełnienie i wypisanie zawartości kontenera `SlowMap`. Na wyjściu widać też skuteczność wywołania `get()`.

Ćwiczenie 15. Powtórz ćwiczenie 13., stosując kontener `SlowMap` (1).

Ćwiczenie 16. Zastosuj testy z programu *Maps.java* do kontenera `SlowMap` i sprawdź, czy kontener przechodzi je pomyślnie. Popraw wykryte niedomagania `SlowMap` (7).

Ćwiczenie 17. Zaimplementuj w klasie `SlowMap` resztę interfejsu `Map` (2).

Ćwiczenie 18. Wzorując się na programie *SlowMap.java*, zaproponuj implementację kontenera `SlowSet` (implementacji interfejsu `Set`) (3).

Haszowanie a szybkość

Program *SlowMap.java* pokazuje, że stworzenie nowego typu `Map` nie jest aż tak trudne. Jednak zgodnie z nazwą `SlowMap` klasa nie jest zbyt szybka, toteż nie należy jej używać, jeżeli jest wobec niej jakaś alternatywa. Cały problem leży w wyszukiwaniu klucza — klucze nie są przechowywane w sposób uporządkowany, stąd stosowane jest zwykłe wyszukiwanie liniowe, będące najwolniejszym ze sposobów poszukiwania wartości w kolekcji.

Najważniejsza zaleta haszowania to właśnie szybkość: haszowanie pozwala na znacznie szybsze szukanie. Ponieważ wąskim gardłem jest właśnie wyszukiwanie klucza, to jednym z rozwiązań mogłoby być przechowywanie kluczy posortowanych, a potem wyszukiwanie ich metodą `Collections.binarySearch()` (przećwiczymy to niebawem).

Haszowanie idzie dalej, zakładając, że najlepsze, co można zrobić, to zapisać *gdzieś* klucz tak, by można go szybko odnaleźć. Najszybszą strukturą do przechowywania zbioru elementów jest tablica, a więc użyjemy jej do reprezentacji informacji o kluczach (uwaga — „informacji o kluczach", a nie samych kluczy). Ale ponieważ tablice po alokacji nie mogą zmieniać rozmiaru, mamy problem: chcemy mieć możliwość zapisania dowolnej liczby wartości w odwzorowaniu `Map`, ale jak to zrobić, gdy liczba kluczy jest ograniczona rozmiarem tablicy?

Odpowiedź jest związana z tym, że tablica nie przechowuje kluczy. Z obiektu klucza wydobędziemy liczbę, która będzie indeksem tablicy. Liczba ta to *kod haszujący* zwracany przez metodę `hashCode()` (w żargonie informatycznym nazywa się ją *funkcją haszującą* albo *funkcją skrótu*), zdefiniowaną w klasie `Object` i oczywiście przesłoniętą przez naszą klasę.

Aby przezwyciężyć kłopoty z tablicami o ustalonych rozmiarach, zakładamy, że więcej niż jeden klucz może dawać ten sam indeks. Zatem oznacza to, że mogą wystąpić *kolizje*. Z tego też względu nie ma znaczenia, jak duża jest tablica, bo i tak każdy obiekt-klucz gdzieś w niej wyląduje.

Wyszukiwanie wartości rozpoczyna się od obliczenia wartości funkcji haszującej i wykorzystania jej jako indeksu tablicy. Jeśli możemy zapewnić, że nie będzie kolizji (co byłoby możliwe w przypadku ustalonej liczby wartości), to mielibyśmy *idealną funkcję haszującą*, ale jest to przypadek szczególny[8]. We wszystkich innych sytuacjach kolizje są obsługiwane przez *zewnętrzne wiązania* — komórki tablicy nie wskazują bezpośrednio wartości, lecz listę wartości. Wartości są odnajdywane w sposób liniowy z wykorzystaniem metody `equals()`. Oczywiście ten etap wyszukiwania jest znacznie wolniejszy, lecz gdy funkcja haszująca jest dobra, to w jednej komórce będziemy mieli co najwyżej kilka wartości. Zatem zamiast szukać po całej liście, szybko skaczemy do komórki tabeli, gdzie trzeba porównać tylko kilka pozycji, i już mamy szukaną wartość. Dzięki tej metodzie implementacja `HashMap` jest tak szybka.

[8] Idealna funkcja haszująca jest implementowana w kontenerach `EnumMap` i `EnumSet` biblioteki kontenerów Javy SE5, ponieważ typ wyliczeniowy definiuje ograniczoną, ustaloną liczbę wartości. Zobacz rozdział „Typy wyliczeniowe".

Znając podstawy haszowania, możemy już zaimplementować proste odwzorowanie z haszowaniem:

```
// containers/SimpleHashMap.java
// Implementacja interfejsu Map z haszowaniem.
import java.util.*;
import net.mindview.util.*;

public class SimpleHashMap<K,V> extends AbstractMap<K,V> {
  // Jako rozmiar tablicy haszującej wybierz liczbę
  // pierwszą -- osiągniesz jednolity rozkład wartości:
  static final int SIZE = 997;
  // Nie można utworzyć fizycznej tablicy typu ogólnionego,
  // ale można rzutować w górę na taką tablicę:
  @SuppressWarnings("unchecked")
  LinkedList<MapEntry<K,V>>[] buckets =
    new LinkedList[SIZE];
  public V put(K key, V value) {
    V oldValue = null;
    int index = Math.abs(key.hashCode()) % SIZE;
    if(buckets[index] == null)
      buckets[index] = new LinkedList<MapEntry<K,V>>();
    LinkedList<MapEntry<K,V>> bucket = buckets[index];
    MapEntry<K,V> pair = new MapEntry<K,V>(key, value);
    boolean found = false;
    ListIterator<MapEntry<K,V>> it = bucket.listIterator();
    while(it.hasNext()) {
      MapEntry<K,V> iPair = it.next();
      if(iPair.getKey().equals(key)) {
        oldValue = iPair.getValue();
        it.set(pair); // Zastąpienie starego nowym
        found = true;
        break;
      }
    }
    if(!found)
      buckets[index].add(pair);
    return oldValue;
  }
  public V get(Object key) {
    int index = Math.abs(key.hashCode()) % SIZE;
    if(buckets[index] == null) return null;
    for(MapEntry<K,V> iPair : buckets[index])
      if(iPair.getKey().equals(key))
        return iPair.getValue();
    return null;
  }
  public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> set= new HashSet<Map.Entry<K,V>>();
    for(LinkedList<MapEntry<K,V>> bucket : buckets) {
      if(bucket == null) continue;
      for(MapEntry<K,V> mpair : bucket)
        set.add(mpair);
    }
    return set;
  }
  public static void main(String[] args) {
    SimpleHashMap<String,String> m =
      new SimpleHashMap<String,String>();
```

```
    m.putAll(Countries.capitals(25));
    System.out.println(m);
    System.out.println(m.get("ERITREA"));
    System.out.println(m.entrySet());
  }
} /* Output:
{CAMEROON=Yaounde, CONGO=Brazzaville, CHAD=N'djamena, COTE D'IVOIR (IVORY
COAST)=Yamoussoukro, CENTRAL AFRICAN REPUBLIC=Bangui, GUINEA=Conakry,
BOTSWANA=Gaberone, BISSAU=Bissau, EGYPT=Cairo, ANGOLA=Luanda, BURKINA
FASO=Ouagadougou, ERITREA=Asmara, THE GAMBIA=Banjul, KENYA=Nairobi, GABON=Libreville,
CAPE VERDE=Praia, ALGERIA=Algiers, COMOROS=Moroni, EQUATORIAL GUINEA=Malabo,
BURUNDI=Bujumbura, BENIN=Porto-Novo, BULGARIA=Sofia, GHANA=Accra, DJIBOUTI=Dijibouti,
ETHIOPIA=Addis Ababa}
Asmara
[CAMEROON=Yaounde, CONGO=Brazzaville, CHAD=N'djamena, COTE D'IVOIR (IVORY
COAST)=Yamoussoukro, CENTRAL AFRICAN REPUBLIC=Bangui, GUINEA=Conakry,
BOTSWANA=Gaberone, BISSAU=Bissau, EGYPT=Cairo, ANGOLA=Luanda, BURKINA
FASO=Ouagadougou, ERITREA=Asmara, THE GAMBIA=Banjul, KENYA=Nairobi, GABON=Libreville,
CAPE VERDE=Praia, ALGERIA=Algiers, COMOROS=Moroni, EQUATORIAL GUINEA=Malabo,
BURUNDI=Bujumbura, BENIN=Porto-Novo, BULGARIA=Sofia, GHANA=Accra, DJIBOUTI=Dijibouti,
ETHIOPIA=Addis Ababa]
*///:~
```

Z uwagi na to, że „komórki" tablicy haszującej są często nazywane *kubełkami* (ang. *buckets*), referencja reprezentująca rzeczywistą tablicę otrzymała nazwę `buckets`. Aby uzyskać równomierny rozkład, liczba kubełków jest liczbą pierwszą[9]. Zauważ, że jest to tablica elementów typu `LinkedList`, co automatycznie zabezpiecza nas przed kolizjami — każda nowa pozycja jest zwyczajnie dodawana na koniec listy. Choć Java nie pozwala na tworzenie tablic typów uogólnionych, dopuszcza utworzenie *referencji* takiej tablicy. Aby zapobiec dodatkowemu rzutowaniu w górę, w dalszej części kodu zadbaliśmy więc o zadeklarowanie odpowiedniej referencji i rzutowanie typu tablicy.

W metodzie `put()` następuje przede wszystkim wywołanie metody `hashCode()` dla klucza; jej wynik sprowadzany jest do liczby dodatniej. Następnie, dzięki operacji modulo, liczba ta jest dopasowywana do rozmiaru tablicy `buckets`. Jeśli otrzymana pozycja nie jest zajęta (czyli `null`), to tworzony jest nowy obiekt `LinkedList` do przechowywania obiektów kierowanych do tego kubełka. Jednak normalnie trzeba przejrzeć listę i sprawdzić, czy para już się na niej znajduje, a jeśli tak, to przypisać nową wartość w miejsce istniejącej, a ją z kolei przypisać do `oldValue`. Znacznik `found` służy do informowania, czy para klucz-wartość została znaleziona — jeżeli nie, to nowa para jest dołączana na koniec listy.

Metoda `get()` oblicza indeks w tablicy `buckets` identycznie jak metoda `put()` (to bardzo ważne, aby obie tak samo obliczały kody haszujące, a tym samym trafiały do tych samych komórek). Jeśli pod obliczonym indeksem znajduje się już lista, zostaje ona przeszukana.

[9] Jak się okazuje, rozwiązanie, w którym liczba kubełków jest liczbą pierwszą, nie jest optymalne. Dlatego też w najnowszych implementacjach Javy liczba ta jest określana jako potęga liczby 2 (przewagę tego rozwiązanie potwierdziły wyczerpujące testy). Dzielenie i określanie reszty z dzielenia jest najwolniejszą operacją wykonywaną przez aktualnie używane procesory. W przypadku gdy liczba kubełków jest potęgą dwójki, zamiast dzielenia można wykorzystać maskowanie. Metoda `get()` jest zdecydowanie najczęściej używana, przez co jej procentowy udział w kosztach wykorzystania obiektu jest znaczny. Określając liczbę kubełków jako potęgę dwójki, można wyeliminować ten problem (jednak może to mieć wpływ na działanie niektórych implementacji metody `hashCode()`).

Pamiętaj, że przedstawiona implementacja nie została dostrojona pod kątem wydajności; miała jedynie pokazać ogólną zasadę działania haszujących kontenerów asocjacyjnych. Implementację zoptymalizowaną pod kątem wydajności znajdziesz, zaglądając do kodu źródłowego klasy `java.util.HashMap`. Ponadto — dla uproszczenia — klasa `SimpleHashMap` wykorzystuje w metodzie `entrySet()` podejście znane z implementacji `SlowMap`, które jako nazbyt uproszczone, nie nadaje się do zastosowań ogólnych — `SimpleHashMap` również ma zastosowanie jedynie ilustracyjne.

Ćwiczenie 19. Powtórz ćwiczenie 13., stosując kontener `SimpleHashMap` (1).

Ćwiczenie 20. Zmodyfikuj klasę `SimpleHashMap` tak, aby informowała o kolizjach kodów haszujących i przetestuj nową wersję ze zdublowanym zbiorem danych (wymuszającym kolizje) (3).

Ćwiczenie 21. Zmodyfikuj klasę `SimpleHashMap` tak, aby informowała o liczbie „sondowań" potrzebnych w przypadku poszczególnych kolizji; klasa ma informować, ile razy trzeba było wywołać metodę `next()` iteratora przeglądającego listy `LinkedList` poszczególnych komórek w poszukiwaniu wartości (2).

Ćwiczenie 22. Zaimplementuj dla klasy `SimpleHashMap` metody `remove()` i `clear()` (4).

Ćwiczenie 23. Zaimplementuj dla klasy `SimpleHashMap` pozostałe metody interfejsu `Map` (3).

Ćwiczenie 24. Naśladując program *SimpleHashMap.java*, zaproponuj i przetestuj implementację haszowanego zbioru `SimpleHashSet` (5).

Ćwiczenie 25. Zamiast dla każdego kubełka wykorzystywać iterator `ListIterator`, zmodyfikuj klasę `MapEntry` tak, aby stanowiła samodzielną listę (każdy egzemplarz `MapEntry` powinien posiadać odnośnik do następnego obiektu `MapEntry`). Zmodyfikuj resztę kodu w *SimpleHashMap.java* pod kątem obsługi nowej struktury `MapEntry` (6).

Przesłonięcie metody hashCode()

Teraz, kiedy już wiesz, na czym polega idea haszowania, napisanie własnej metody `hashCode()` ma więcej sensu.

Przede wszystkim nie trzeba kontrolować tworzenia rzeczywistej wartości stosowanej do indeksowania tablicy. Jest ona zależna od pojemności konkretnego obiektu `HashMap`, która zmienia się zależnie od liczby elementów w kontenerze i *współczynnika wypełnienia* (zajmiemy się nim później). Wartość zwracana przez naszą metodę `hashCode()` będzie dalej przetwarzana w celu uzyskania indeksu w tablicy (w klasie `SimpleHashMap` obliczenie to po prostu wzięcie wartości modulo rozmiar tabeli).

Najistotniejszym czynnikiem przy tworzeniu metody `hashCode()` jest to, by niezależnie od tego, kiedy jest wywołana, zwracała tę samą wartość dla konkretnego obiektu przy każdym wywołaniu. Kiedy zbudujemy obiekt, który daje jeden kod haszujący podczas operacji `put()` i inny podczas `get()`, nie będziemy w stanie odzyskać tego obiektu. Zatem jeżeli `hashCode()` zależy od tych danych obiektu, które się zmieniają, to użytkownik powinien być świadomy, że zmiana danych poskutkuje wyliczeniem innego klucza przy różnych wywołaniach funkcji haszującej.

W dodatku prawdopodobnie *nie* będziesz chciał tworzyć metody `hashCode()` bazującej na unikatowej informacji obiektu. Szczególnie `this` jest złą wartością, ponieważ nie można stworzyć nowego klucza pasującego do tego użytego podczas zamieszczania pierwotnej pary klucz-wartość. Jest to problem, który pojawił się w programie *SpringDetector.java*, gdyż domyślna implementacja metody `hashCode()` *używa* właśnie adresu obiektu. A zatem chcemy używać tych danych obiektu, które go identyfikują w sensowny sposób.

Jeden z przykładów widać w klasie `String`. Łańcuchy tekstowe mają wyjątkową właściwość — jeżeli w programie jest kilka obiektów klasy `String`, zawierających identyczny tekst, to wszystkie takie obiekty odnoszą się do tej samej pamięci. Zatem sensowne jest, że kod haszujący zwracany przez dwie odrębne instancje `new String("Witaj")` powinien być identyczny. Możesz to zaobserwować, uruchamiając następujący program:

```
// containers/StringHashCode.java

public class StringHashCode {
  public static void main(String[] args) {
    String[] hellos = "Witaj Witaj".split(" ");
    System.out.println(hellos[0].hashCode());
    System.out.println(hellos[1].hashCode());
  }
} /* Output: (Sample)
83588971
83588971
*///:~
```

Jak wyraźnie widać, metoda `hashCode()` dla klasy `String` operuje na zawartości łańcucha znaków.

Zatem, aby metoda `hashCode()` była skuteczna, musi być szybka i powinna generować wartości na podstawie zawartości obiektu. Pamiętaj, że wartość ta nie musi być unikatowa — powinieneś pójść raczej w kierunku szybkości niż jednoznaczności — ale stosując `hashCode()` i `equals()` identyczność obiektów musi być całkowicie pewna.

Ponieważ zanim indeks komórki zostanie obliczony, kod haszujący jest dalej przetwarzany, to zakres wartości nie jest istotny, kod powinien być tylko typu `int`.

Jest jeszcze jeden czynnik: dobra metoda `hashCode()` powinna dawać równomierny rozkład wartości. Jeśli wartości mają tendencję do grupowania, to `HashMap` lub też `HashSet` będą posiadały bardziej wypełnione pewne obszary i nie będą tak szybkie, jak mogłyby być dzięki równomiernie rozkładającej funkcji haszującej.

W książce *Effective Java™ Programming Language Guide* (wydanej w 2001 roku przez wydawnictwo Addison-Wesley) Joshua Bloch podaje prosty sposób na stworzenie poprawnej metody `hashCode()`:

1. Zapisz pewną stałą wartość całkowitą (na przykład: 17) w zmiennej `result`.
2. Wyznacz kod haszujący c dla każdego znaczącego pola `f` obiektu (czyli pola, którego wartość powinna być uwzględniana przy wyznaczaniu kodu haszującego); kod ten powinien być liczbą typu `int`:

Typ pola	Obliczenie
boolean	`c = (f ? 0 : 1)`
byte, char, short lub int	`c = (int)f`
long	`c = (int)(f ^ (f>>>32))`
float	`c = Float.floatToIntBits(f);`
double	`long l = Double.doubleToLongBits(f);` `c = (int)(l ^ (l >>> 32))`
Object, gdzie metoda equals() wywołuje equals() dla danego pola	`c = f.hashCode()`
tablica	Dla każdego elementu tablicy należy zastosować powyższe reguły.

3. Połącz wszystkie wyznaczone kody haszujące, posługując się wzorem:

```
result = 37 * result + c
```

4. Zwróć wartość zmiennej result.

5. Przeanalizuj powstałą w ten sposób metodę hashCode() i upewnij się, że różne instancje obiektu będą miały ten sam kod haszujący.

Oto przykład, który uwzględnia powyższe wskazówki:

```
//: containers/CountedString.java
// Dobra funkcja haszująca hashCode().
import java.util.*;
import static net.mindview.util.Print.*;

public class CountedString {
  private static List<String> created =
    new ArrayList<String>();
  private String s;
  private int id = 0;
  public CountedString(String str) {
    s = str;
    created.add(s);
    // id to łączna liczba egzemplarzy
    // ciągu w użyciu CountedString:
    for(String s2 : created)
      if(s2.equals(s))
        id++;
  }
  public String toString() {
    return "Ciąg: " + s + " id: " + id +
      " hashCode(): " + hashCode();
  }
  public int hashCode() {
    // Wersja uproszczona:
    // return s.hashCode() * id;
    // Wersja na bazie porady J. Blocha:
    int result = 17;
    result = 37 * result + s.hashCode();
    result = 37 * result + id;
    return result;
```

```
  }
  public boolean equals(Object o) {
    return o instanceof CountedString &&
      s.equals(((CountedString)o).s) &&
      id == ((CountedString)o).id;
  }
  public static void main(String[] args) {
    Map<CountedString,Integer> map =
      new HashMap<CountedString,Integer>();
    CountedString[] cs = new CountedString[5];
    for(int i = 0; i < cs.length; i++) {
      cs[i] = new CountedString("hi");
      map.put(cs[i], i); // Pakowanie int -> Integer
    }
    print(map);
    for(CountedString cstring : cs) {
      print("Szukanie " + cstring);
      print(map.get(cstring));
    }
  }
} /* Output: (Sample)
{Ciąg: hi id: 4 hashCode(): 146450=3, Ciąg: hi id: 1 hashCode(): 146447=0, Ciąg: hi id: 3 hashCode():
146449=2, Ciąg: hi id: 5 hashCode(): 146451=4, Ciąg: hi id: 2 hashCode(): 146448=1}
Szukanie Ciąg: hi id: 1 hashCode(): 146447
0
Szukanie Ciąg: hi id: 2 hashCode(): 146448
1
Szukanie Ciąg: hi id: 3 hashCode(): 146449
2
Szukanie Ciąg: hi id: 4 hashCode(): 146450
3
Szukanie Ciąg: hi id: 5 hashCode(): 146451
4
*///:~
```

Klasa `CountedString` zawiera zmienną typu `String` oraz `id` reprezentujące liczbę obiektów `CountedString` zawierających identyczne łańcuchy tekstowe. Zliczanie jest realizowane w konstruktorze poprzez iterowanie statycznej `ArrayList`, w której umieszczone są wszystkie łańcuchy tekstowe.

Obie metody — `hashCode()` i `equals()` — dają wynik na podstawie obu pól; gdyby opierały się tylko na samej zmiennej łańcuchowej albo tylko na `id`, to mielibyśmy wielokrotne dopasowania dla różnych wartości.

W metodzie głównej tworzona jest grupa obiektów z tym samym tekstem, aby pokazać, że duplikaty zyskują unikatową wartość dzięki licznikowi `id`. `HashMap` jest wypisywana, co pozwala nam zobaczyć, jak wewnętrznie wygląda przechowywanie elementów (niedostrzegalny porządek). Następnie wyszukiwany jest każdy z kluczy, pokazując tym samym poprawność działania mechanizmu wyszukiwania.

W ramach następnego przykładu zajmiemy się klasą `Individual`, wykorzystywaną w roli klasy bazowej w bibliotece `typeinfo.pets` powołanej do życia w rozdziale „Informacje o typach". Klasa `Individual` była wykorzystywana w tamtym rozdziale, ale jej definicja została opóźniona aż do teraz, kiedy możesz w pełni ogarnąć sens jej implementacji:

```
// typeinfo/pets/Individual.java
package typeinfo.pets;

public class Individual implements Comparable<Individual> {
  private static long counter = 0;
  private final long id = counter++;
  private String name;
  public Individual(String name) { this.name = name; }
  // pole 'name' jest opcjonalne:
  public Individual() {}
  public String toString() {
    return getClass().getSimpleName() +
      (name == null ? "" : " " + name);
  }
  public long id() { return id; }
  public boolean equals(Object o) {
    return o instanceof Individual &&
      id == ((Individual)o).id;
  }
  public int hashCode() {
    int result = 17;
    if(name != null)
      result = 37 * result + name.hashCode();
    result = 37 * result + (int)id;
    return result;
  }
  public int compareTo(Individual arg) {
    // Porównywanie po nazwie podtypu:
    String first = getClass().getSimpleName();
    String argFirst = arg.getClass().getSimpleName();
    int firstCompare = first.compareTo(argFirst);
    if(firstCompare != 0)
    return firstCompare;
    if(name != null && arg.name != null) {
      int secondCompare = name.compareTo(arg.name);
      if(secondCompare != 0)
        return secondCompare;
    }
    return (arg.id < id ? -1 : (arg.id == id ? 0 : 1));
  }
} ///:~
```

Metoda `compareTo()` realizuje hierarchię porównań, porządkując sekwencję według typu, potem według nazwy (`name`) — o ile została określona — a następnie według kolejności tworzenia egzemplarzy. Oto przykład ilustrujący efekty takiego porównania:

```
// containers/IndividualTest.java
import holding.MapOfList;
import typeinfo.pets.*;
import java.util.*;

public class IndividualTest {
  public static void main(String[] args) {
    Set<Individual> pets = new TreeSet<Individual>();
    for(List<? extends Pet> lp :
        MapOfList.petPeople.values())
      for(Pet p : lp)
        pets.add(p);
```

```
    System.out.println(pets);
  }
} /* Output:
[Cat Mala Lu, Cat Pinkola, Cat Stefan vel Czarny Łobuz, Cat Szoruś, Cymric Molly, Dog Marten, Mutt
Spot, Pug Lolo vel Leonard Moppsen, Rat Gonek, Rat Kolka, Rat Lelek]
*///:~
```

Ponieważ wszystkie zwierzaki mają imiona, są porządkowane najpierw według typu, potem według imienia (nazwy egzemplarza w obrębie typu).

Tworzenie metod `hashCode()` oraz `equals()` dla nowych klas może być kłopotliwe. Narzędzia, które mogą w tym pomóc, można znaleźć w projekcie *Jakarta Commons* prowadzonym przez organizację Apache Software Foundation, pod adresem *http://jakarta.apache.org/commons/* w dziale „lang". (W projekcie *Jakarta Commons* można znaleźć także wiele innych, potencjalnie przydatnych bibliotek; wydaje się, że jest on odpowiedzią społeczności programistów korzystających z Javy na witrynę *www.boost.org* udostępniającą kody bibliotek w języku C++).

Ćwiczenie 26. Dodaj do klasy `CountedString` pole typu `char` inicjalizowane z poziomu konstruktora i zmodyfikuj metody `hashCode()` i `equals()` klasy tak, aby uwzględniały wartość nowego pola (2).

Ćwiczenie 27. Zmodyfikuj metodę `hashCode()` w przykładzie *CountedString.java* poprzez usunięcie mnożenia przez wartość `id` i wykaż, że `CountedString` wciąż działa jako klucz. Jaki problem pojawia się w tym rozwiązaniu (3)?

Ćwiczenie 28. Zmodyfikuj program *net/mindview/util/Tuple.java* tak, aby powstała klasa ogólnego przeznaczenia, wyposażona w metody `hashCode()` i `equals()` oraz implementująca interfejs `Comparable` (4)?

Wybór implementacji

Wiesz już, że tak naprawdę są tylko cztery komponenty kontenerowe, `Map`, `List`, `Set` oraz `Queue`, i niekiedy po kilka implementacji każdego z tych interfejsów. Jeśli zatem potrzebne będzie wykorzystanie funkcji oferowanych przez konkretny interfejs, jak zdecydować, którą implementację wykorzystać?

Każda z implementacji ma swoje specyficzne właściwości, zalety i wady. Przykładowo, jak widać na diagramie, „cechą" `Hashtable`, `Vector` i `Stack` jest to, że są klasami przestarzałymi, więc nie popsują starszego kodu. Z drugiej strony, lepiej ich nie używać w nowych programach.

Typy kolejek (implementacji interfejsu `Queue`) różnią się w zasadzie jedynie sposobem przyjmowania i udostępniania elementów (istota tych rozróżnień wyjaśni się w rozdziale „Współbieżność").

Różnice między innymi kontenerami często sprowadzają się do tego, jak są zrealizowane, czyli jaką strukturę fizycznie implementuje wybrany przez nas interfejs. Znaczy to, że na przykład `ArrayList` i `LinkedList`, implementując interfejs `List`, zapewnią, iż niezależnie od

tego, której z nich użyjemy w programie, to uzyskamy ten sam wynik. Jednak `ArrayList` jest realizowana przez tablicę, podczas gdy `LinkedList` jako normalna lista dwukierunkowa — z węzłami jako osobnymi obiektami, zawierającymi wartość oraz wskazania na poprzedni i następny element na liście. Z tego powodu, jeśli chcielibyśmy wielokrotnie wstawiać lub usuwać elementy ze środka listy, to `LinkedList` byłaby właściwym wyborem (`LinkedList` ma również dodatkowe właściwości ustalone w `AbstractSequentialList`). Jeżeli nie, to zasadniczo szybsza jest `ArrayList`.

Kolejny przykład: zbiór `Set` może być zaimplementowany jako `TreeSet`, `HashSet` lub `LinkedHashSet`[10]. Każdy z nich cechuje się odmiennym zachowaniem: `HashSet` jest przeznaczony do typowych zastosowań i ma dużą szybkość wyszukiwania, `LinkedHashSet` przechowuje pary w kolejności, w jakiej były dodawane, z kolei `TreeSet` bazuje na `TreeMap` i został zaprojektowany w celu udostępnienia zbioru, którego elementy zawsze będą posortowane. Pomysł polega na tym, aby w zależności od zachowania, jakie ma mieć zbiór, można było wybrać odpowiednią implementację.

Niekiedy różne implementacje danego kontenera udostępniają wspólne operacje, ale implementują je z różną wydajnością. W takim przypadku wybór implementacji należy uzależnić od częstotliwości wykonywania poszczególnych operacji i wymaganej szybkości działania. Jedną z metod rozpoznania i wybrania odpowiedniego kontenera jest w takim przypadku test wydajności.

Infrastruktura testowa

Aby uniknąć powielania kodu i zapewnić spójność procedur testowych, postanowiłem wyodrębnić coś w rodzaju szkieletu platformy testowej. Prezentowany tu kod ustala klasę bazową, na podstawie której można tworzyć listy anonimowych klas wewnętrznych, po jednej dla każdego testu. Każda z owych klas wewnętrznych pełni rolę części ogólnej procedury testowej. Pozwala to na proste dodawanie i usuwanie nowych rodzajów testów.

Mamy tu do czynienia z kolejną realizacją wzorca projektowego *Template Method*, czyli „szablon metody". Choć zgodnie z założeniami wzorca projektowego dla każdego testu przesłaniamy metodę `Test.test()`, rdzeń kodu (jego część niezmienna) został wyodrębniony do klasy `Tester`[11]. Typ testowanego kontenera jest reprezentowany parametrem typowym `C`.

```
// : containers/Test.java
// Szkielet testów wydajnościowych kontenerów.

public abstract class Test<C> {
  String name;
  public Test(String name) { this.name = name; }
  // Metoda przesłaniana w poszczególnych testach.
  // Zwraca osiągniętą liczbę powtórzeń testu.
  abstract int test(C container, TestParam tp);
} ///:~
```

[10] Albo jako `EnumSet` tudzież `CopyOnWriteArraySet` w roli przypadków specjalnych. Wiedza o dodatkowych specjalizowanych implementacjach rozmaitych interfejsów kontenerowych jest jak najbardziej pożądana, ale w tym rozdziale koncentruję się na zastosowaniach — i implementacjach — ogólnego przeznaczenia.

[11] Przy rozpracowaniu uogólnień do tego przykładu pomagał mi Krzysztof Sobolewski.

Każdy egzemplarz klasy `Test` przechowuje nazwę testu. Wywołanie metody `test()` wymaga przekazania kontenera przeznaczonego do testowania oraz obiektu „posłańca" czy też „obiektu transferu danych" przekazującego rozmaite parametry danego testu. Do owych parametrów należy choćby `size`, regulujący liczbę elementów kontenera, oraz `loops`, wyznaczający liczbę powtórzeń danego testu. Parametry te nie muszą być koniecznie wykorzystywane we wszystkich testach.

Każdy kontener przejdzie przez sekwencję wywołań metody `test()`, za każdym razem z innym zestawem parametrów `TestParam`; klasa `TestParam` zawiera więc statyczną metodę `array()`, ułatwiającą tworzenie tablic obiektów tej klasy. Pierwsza wersja metody `array()` przyjmuje zmienną listę argumentów z argumentami regulującymi wartości `size` i `loops`, druga wersja przyjmuje taką samą listę, z tym że wartości są na niej reprezentowane ciągami znaków (obiektami klasy `String`) — tę wersję można wykorzystać do przetwarzania argumentów wiersza wywołania programu:

```
//: containers/TestParam.java
// Obiekt parametrów testu.

public class TestParam {
  public final int size;
  public final int loops;
  public TestParam(int size, int loops) {
    this.size = size;
    this.loops = loops;
  }
  // Metoda tworząca tablicę obiektów TestParam
  // na podstawie listy argumentów:
  public static TestParam[] array(int... values) {
    int size = values.length/2;
    TestParam[] result = new TestParam[size];
    int n = 0;
    for(int i = 0; i < size; i++)
      result[i] = new TestParam(values[n++], values[n++]);
    return result;
  }
  // Konwersja tablicy ciągów znaków na tablicę obiektów TestParam:
  public static TestParam[] array(String[] values) {
    int[] vals = new int[values.length];
    for(int i = 0; i < vals.length; i++)
      vals[i] = Integer.decode(values[i]);
    return array(vals);
  }
} ///:~
```

Aby skorzystać z infrastruktury testowej, należy przekazać testowany kontener wraz z listą obiektów `Test` do metody `Tester.run()` (są one przeciążonymi uogólnionymi metodami pomocniczymi, co pozwala zredukować ilość kodu potrzebnego do ich stosowania). Metoda `Tester.run()` wywoła odpowiedni konstruktor przeciążony, a potem metodę `timedTest()`, która uruchomi testy z listy testów skojarzonej z danym kontenerem. Metoda `timedTest()` uruchamia test dla wszystkich obiektów parametrów (`TestParam`) z listy `paramList`. Ponieważ obiekt `paramList` jest inicjalizowany na podstawie statycznej tablicy `defaultParams`, można wygodnie zmieniać parametry wszystkich testów, zmieniając przypisania tablicy `defaultParams`. Można też zmieniać parametry dla konkretnego testu, przekazując własny egzemplarz `paramList` dla tego testu.

```
// : containers/Tester.java
// Aplikuje obiekty Test do list różnych kontenerów.
import java.util.*;

public class Tester<C> {
  public static int fieldWidth = 8;
  public static TestParam[] defaultParams= TestParam.array(
    10, 5000, 100, 5000, 1000, 5000, 10000, 500);
  // Przesłonić tę metodę w celu zmiany inicjalizacji wstępnej:
  protected C initialize(int size) { return container; }
  protected C container;
  private String headline = "";
  private List<Test<C>> tests;
  private static String stringField() {
    return "%" + fieldWidth + "s";
  }
  private static String numberField() {
    return "%" + fieldWidth + "d";
  }
  private static int sizeWidth = 8;
  private static String sizeField = "%" + sizeWidth + "s";
  private TestParam[] paramList = defaultParams;
  public Tester(C container, List<Test<C>> tests) {
    this.container = container;
    this.tests = tests;
    if(container != null)
      headline = container.getClass().getSimpleName();
  }
  public Tester(C container, List<Test<C>> tests,
      TestParam[] paramList) {
    this(container, tests);
    this.paramList = paramList;
  }
  public void setHeadline(String newHeadline) {
    headline = newHeadline;
  }
  // Pomocnicze metody uogólnione:
  public static <C> void run(C cntnr, List<Test<C>> tests){
    new Tester<C>(cntnr, tests).timedTest();
  }
  public static <C> void run(C cntnr,
      List<Test<C>> tests, TestParam[] paramList) {
    new Tester<C>(cntnr, tests, paramList).timedTest();
  }
  private void displayHeader() {
    // Obliczenie długości wypełnienia znakami '-':
    int width = fieldWidth * tests.size() + sizeWidth;
    int dashLength = width - headline.length() - 1;
    StringBuilder head = new StringBuilder(width);
    for(int i = 0; i < dashLength/2; i++)
      head.append('-');
    head.append(' ');
    head.append(headline);
    head.append(' ');
    for(int i = 0; i < dashLength/2; i++)
      head.append('-');
    System.out.println(head);
    // Wypisanie kolumny nagłówkowej:
```

```
    System.out.format(sizeField, "rozmiar");
    for(Test test : tests)
      System.out.format(stringField(), test.name);
    System.out.println();
  }
  // Uruchomienie testów dla danego kontenera:
  public void timedTest() {
    displayHeader();
    for(TestParam param : paramList) {
      System.out.format(sizeField, param.size);
      for(Test<C> test : tests) {
        C kontainer = initialize(param.size);
        long start = System.nanoTime();
        // Wywołanie metody przesłoniętej:
        int reps = test.test(kontainer, param);
        long duration = System.nanoTime() - start;
        long timePerRep = duration / reps; // W nanosekundach
        System.out.format(numberField(), timePerRep);
      }
      System.out.println();
    }
  }
} ///:~
```

Metody `stringField()` i `numberField()` zwracają sformatowane ciągi znaków wykorzystywane przy wypisywaniu wyników. Standardową szerokość formatowania można zmienić, modyfikując statyczne pole `fieldWidth`. Metoda `displayHeader()` formatuje i wypisuje informacje nagłówka zestawienia wyników każdego testu.

Tam, gdzie potrzebna jest specjalna inicjalizacja, należy przesłonić metodę `initialize()`. Metoda ta zwraca zainicjalizowany (wypełniony) kontener odpowiedniego rozmiaru — można tu albo zmodyfikować istniejący obiekt kontenera, albo utworzyć całkiem nowy. W metodzie `test()` wyniki są przechwytywane do lokalnej referencji o nazwie `kontainer`, co pozwala na zastąpienie przechowywanej składowej `container` zupełnie innym kontenerem zainicjalizowanym.

Wartość zwracana z każdego wywołania `Test.test()` musi reprezentować liczbę operacji wykonanych w ramach testu, co pozwala na obliczenie czasu trwania poszczególnych operacji (w nanosekundach). Trzeba mieć świadomość, że wywołania `System.nanoTime()` zwracają zazwyczaj wartości o ziarnistości większej od jedności (dokładność pomiaru czasu jest zależna od komputera i systemu operacyjnego), co w pewnym stopniu zakłóca wyniki.

Wyniki testów będą różne na różnych komputerach; ich prezentacja ma więc jedynie na celu określenie względnej wydajności poszczególnych kontenerów.

Wybieranie pomiędzy listami

Oto test wydajności najważniejszych operacji właściwych dla implementacji interfejsu `List`. Dla porównania test obejmuje również podstawowe operacje na kolejkach (`Queue`). Do testu każdej klasy kontenerowej tworzone są dwie listy testów. W tym przypadku operacje właściwe dla interfejsu `Queue` stosowane są tylko do implementacji `LinkedList`.

```
// : containers/ListPerformance.java
// Ilustracja wydajności różnych implementacji List.
// {Args: 100 500} Mała liczba skraca czas dla systemu kompilacji
import java.util.*;
import net.mindview.util.*;

public class ListPerformance {
  static Random rand = new Random();
  static int reps = 1000;
  static List<Test<List<Integer>>> tests =
    new ArrayList<Test<List<Integer>>>();
  static List<Test<LinkedList<Integer>>> qTests =
    new ArrayList<Test<LinkedList<Integer>>>();
  static {
    tests.add(new Test<List<Integer>>("add") {
      int test(List<Integer> list, TestParam tp) {
        int loops = tp.loops;
        int listSize = tp.size;
        for(int i = 0; i < loops; i++) {
          list.clear();
          for(int j = 0; j < listSize; j++)
            list.add(j);
        }
        return loops * listSize;
      }
    });
    tests.add(new Test<List<Integer>>("get") {
      int test(List<Integer> list, TestParam tp) {
        int loops = tp.loops * reps;
        int listSize = list.size();
        for(int i = 0; i < loops; i++)
          list.get(rand.nextInt(listSize));
        return loops;
      }
    });
    tests.add(new Test<List<Integer>>("set") {
      int test(List<Integer> list, TestParam tp) {
        int loops = tp.loops * reps;
        int listSize = list.size();
        for(int i = 0; i < loops; i++)
          list.set(rand.nextInt(listSize), 47);
        return loops;
      }
    });
    tests.add(new Test<List<Integer>>("iteradd") {
      int test(List<Integer> list, TestParam tp) {
        final int LOOPS = 1000000;
        int half = list.size() / 2;
        ListIterator<Integer> it = list.listIterator(half);
        for(int i = 0; i < LOOPS; i++)
          it.add(47);
        return LOOPS;
      }
    });
    tests.add(new Test<List<Integer>>("insert") {
      int test(List<Integer> list, TestParam tp) {
        int loops = tp.loops;
        for(int i = 0; i < loops; i++)
```

```
      list.add(5, 47); // Minimalizacja kosztu dostępu swobodnego
    return loops;
  }
});
tests.add(new Test<List<Integer>>("remove") {
  int test(List<Integer> list, TestParam tp) {
    int loops = tp.loops;
    int size = tp.size;
    for(int i = 0; i < loops; i++) {
      list.clear();
      list.addAll(new CountingIntegerList(size));
      while(list.size() > 5)
        list.remove(5); // Minimalizacja kosztu dostępu swobodnego
    }
    return loops * size;
  }
});
// Testy kolejki:
qTests.add(new Test<LinkedList<Integer>>("addFirst") {
  int test(LinkedList<Integer> list, TestParam tp) {
    int loops = tp.loops;
    int size = tp.size;
    for(int i = 0; i < loops; i++) {
      list.clear();
      for(int j = 0; j < size; j++)
        list.addFirst(47);
    }
    return loops * size;
  }
});
qTests.add(new Test<LinkedList<Integer>>("addLast") {
  int test(LinkedList<Integer> list, TestParam tp) {
    int loops = tp.loops;
    int size = tp.size;
    for(int i = 0; i < loops; i++) {
      list.clear();
      for(int j = 0; j < size; j++)
        list.addLast(47);
    }
    return loops * size;
  }
});
qTests.add(
  new Test<LinkedList<Integer>>("rmFirst") {
    int test(LinkedList<Integer> list, TestParam tp) {
      int loops = tp.loops;
      int size = tp.size;
      for(int i = 0; i < loops; i++) {
        list.clear();
        list.addAll(new CountingIntegerList(size));
        while(list.size() > 0)
          list.removeFirst();
      }
      return loops * size;
    }
  });
qTests.add(new Test<LinkedList<Integer>>("rmLast") {
  int test(LinkedList<Integer> list, TestParam tp) {
```

```
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
          list.clear();
          list.addAll(new CountingIntegerList(size));
          while(list.size() > 0)
            list.removeLast();
        }
        return loops * size;
      }
    });
  }
  static class ListTester extends Tester<List<Integer>> {
    public ListTester(List<Integer> container,
        List<Test<List<Integer>>> tests) {
      super(container, tests);
    }
    // Wypełnienie do pożądanego rozmiaru przed każdym testem:
    @Override protected List<Integer> initialize(int size){
      container.clear();
      container.addAll(new CountingIntegerList(size));
      return container;
    }
    // Metoda pomocnicza:
    public static void run(List<Integer> list,
        List<Test<List<Integer>>> tests) {
      new ListTester(list, tests).timedTest();
    }
  }
  public static void main(String[] args) {
    if(args.length > 0)
      Tester.defaultParams = TestParam.array(args);
    // Na tablicy można przeprowadzić jedynie dwa poniższe testy:
    Tester<List<Integer>> arrayTest =
      new Tester<List<Integer>>(null, tests.subList(1, 3)){
        // Wywoływana przed każdym testem. Tworzy
        // listę bazującą na tablicy o stałym rozmiarze:
        @Override protected
        List<Integer> initialize(int size) {
          Integer[] ia = Generated.array(Integer.class,
            new CountingGenerator.Integer(), size);
          return Arrays.asList(ia);
        }
      };
    arrayTest.setHeadline("Tablica jako List");
    arrayTest.timedTest();
    Tester.defaultParams= TestParam.array(
      10, 5000, 100, 5000, 1000, 1000, 10000, 200);
    if(args.length > 0)
      Tester.defaultParams = TestParam.array(args);
    ListTester.run(new ArrayList<Integer>(), tests);
    ListTester.run(new LinkedList<Integer>(), tests);
    ListTester.run(new Vector<Integer>(), tests);
    Tester.fieldWidth = 12;
    Tester<LinkedList<Integer>> qTest =
      new Tester<LinkedList<Integer>>(
        new LinkedList<Integer>(), qTests);
    qTest.setHeadline("Testy kolejki");
```

```
      qTest.timedTest();
    }
  } /* Output: (Sample)
  --- Tablica jako List ---
  rozmiar    get    set
     10      130    183
    100      130    164
   1000      129    165
  10000      129    165
  ---------- ArrayList ----------
  rozmiar    add    get    set iteradd insert remove
     10      121    139    191    435   3952    446
    100       72    141    191    247   3934    296
   1000       98    141    194    839   2202    923
  10000      122    144    190   6880  14042   7333
  ---------- LinkedList ----------
  rozmiar    add    get    set iteradd insert remove
     10      182    164    198    658    366    262
    100      106    202    230    457    108    201
   1000      133   1289   1353    430    136    239
  10000      172  13648  13187    435    255    239
  ---------- Vector ----------
  rozmiar    add    get    set iteradd insert remove
     10      129    145    187    290   3635    253
    100       72    144    190    263   3691    292
   1000       99    145    193    846   2162    927
  10000      108    145    186   6871  14730   7135
  ---------- Testy kolejki ----------
  rozmiar   addFirst   addLast   rmFirst   rmLast
     10       199       163       251       253
    100        98        92       180       179
   1000        99        93       216       212
  10000       111       109       262       384
  *///:~
```

Każdy test wymaga zaplanowania tak, aby dawał sensowne i znaczące wyniki. Na przykład test „add" (dodawania elementów do kontenera) za każdym razem opróżnia kontener z elementów, po czym wypełnia go elementami do pożądanego rozmiaru. Z tego względu częścią testu jest wywołanie metody `clear()`, która może wpływać na czas wykonania testu, zwłaszcza w przypadku niewielkich kontenerów i krótkich testów. Choć wyniki wydają się sensowne, można zastanowić się nad przekonstruowaniem infrastruktury testowej tak, aby pozwalała na *wyłączenie* procedur przygotowawczych testu (w naszym przypadku taką procedurą byłoby wywołanie metody `clear()`) z pomiaru czasu.

Zauważ, że w ramach każdego testu trzeba pilnie zliczać liczbę powtórzeń, aby potem przekazać ją do wywołującego jako wartość zwracaną z metody `test()` i aby dało się na tej podstawie obliczyć czas trwania pojedynczego przebiegu testu.

Testy „get" i „set" wykorzystują generator liczb losowych do wykonywania swobodnych odwołań do elementów listy. Na wyjściu programu widać, że w przypadku listy bazującej na zwyczajnej tablicy i kontenera `ArrayList` operacje dostępu swobodnego są szybkie, a ich czas w zasadzie nie zależy od rozmiaru kontenera. Z kolei dla implementacji `LinkedList` średni czas realizacji odwołań swobodnych rośnie wraz z wielkością kontenera. Najwyraźniej w zastosowaniach wymagających częstego dostępu swobodnego lepiej korzystać z implementacji bazujących na tablicach.

Test „iteradd" wykorzystuje iterator do wstawiania nowych elementów do środka listy. W przypadku kontenera ArrayList operacja ta okazuje się kosztowną, zwłaszcza dla większych rozmiarów kontenera, za to w implementacji LinkedList ta sama operacja jest względnie szybka i do tego wykonywana w czasie stałym, niezależnie od rozmiaru listy. To logiczne, ponieważ kontener ArrayList musi w ramach operacji wstawiania utworzyć nowy kontener i skopiować do niego wszystkie referencje z pierwotnej listy. Całość dla większych list staje się długotrwała. W przypadku kontenera LinkedList analogiczna operacja, niezależnie od rozmiaru listy, sprowadza się do wstawienia nowego węzła listy pomiędzy dwa istniejące, bez konieczności modyfikowania czy kopiowania pozostałych węzłów listy.

Testy „insert" i „remove" polegają na wstawianiu i usuwaniu elementu na piątej pozycji kontenera (test „add" dodawał element na koniec kontenera). Implementacja LinkedList obsługuje końcowe węzły listy w sposób szczególny, co ma zwiększać efektywność takiej listy, kiedy występuje ona w roli kolejki (implementacji interfejsu Queue). Ale dodawanie elementów do wnętrza listy (czy ich usuwanie stamtąd) wymaga ujęcia również kosztu dostępu swobodnego do elementu, który mierzyliśmy wcześniej. Wybór pozycji o numerze 5 powinien minimalizować udział kosztu dostępu swobodnego przy równoczesnym wyeliminowaniu optymalizacji LinkedList związanych z obsługą samych końcówek listy. Wyniki wyprowadzone na wyjście programu dowodzą, że wstawianie i usuwanie w przypadku implementacji LinkedList jest mało kosztowne i nie zależy od rozmiaru listy; kontener ArrayList wprowadza za to *duży* narzut operacji wstawiania elementu, rosnący wraz z rozmiarem listy.

Wyniki testów kolejki pokazują, że faktycznie implementacja LinkedList optymalizuje operacje dotyczące obu końcówek listy, co ma uzasadnienie z punktu widzenia pożądanej charakterystyki kolejek.

Normalnie test polegałby na wywołaniu metody Tester.run() z kontenerem i listą testów do wykonania. Tu jednak trzeba było przesłonić metodę initialize() tak, aby lista była przed każdym testem opróżniana i ponownie wypełniana — w innym przypadku w trakcie rozmaitych testów stracilibyśmy kontrolę nad rozmiarem listy. Klasa ListTester dziedziczy po klasie bazowej Tester i realizuje pożądaną inicjalizację za pośrednictwem klasy CountingIntegerList. Przesłonięta została również metoda pomocnicza run().

Przy okazji postanowiłem porównać operacje kontenerowe z operacjami tablicowymi (chodziło mi zwłaszcza o porównanie zwyczajnych tablic z implementacją ArrayList). W pierwszym teście wykonywanym w metodzie main() tworzony jest więc specjalny obiekt Test w postaci anonimowej klasy wewnętrznej. Klasa ta przesłania metodę initialize() tak, aby każde wywołanie tworzyło nowy obiekt (ignorując obiekt container — argumentem dla konstruktora tego obiektu jest więc null). Nowy obiekt testu powstaje jako wynik wywołania metod Generated.array() (zobacz rozdział „Tablice") i Arrays.asList(). W tym przypadku można przeprowadzić jedynie dwa testy, bo do kontenera List opartego na tablicy fizycznej nie można wstawiać elementów ani ich z niego usuwać; podzbiór możliwych testów jest wybierany z listy testów metodą List.subList().

W zakresie operacji get() i set() implementacja List bazująca na zwyczajnej tablicy jest tylko nieznacznie szybsza od implementacji ArrayList; w przypadku LinkedList te same operacje są dramatycznie wolniejsze, bo implementacja LinkedList nie została zoptymalizowana pod kątem dostępu swobodnego.

Kontenera `Vector` należałoby w ogóle unikać; jego obecność w bibliotece kontenerów uzasadnia jedynie wzgląd na starszy kod, a działa w tym programie tylko dlatego, że dla osiągnięcia zgodności wprzód kontener `Vector` został zaadaptowany jako implementacja `List`.

Z testów wynika, że najlepszym wyborem domyślnym byłby kontener `ArrayList`, zamieniany ewentualnie na `LinkedList` tam, gdzie potrzebne są dodatkowe funkcje tej implementacji albo gdzie dochodzi do istotnego spowolnienia wynikającego z licznych operacji wstawiania i usuwania elementów we wnętrzu kontenera. W przypadku manipulowania kolekcjami elementów o ustalonej liczności można skorzystać z listy bazującej na tablicy fizycznej (generowanej wywołaniem `Arrays.asList()`) albo nawet ze zwyczajnej tablicy.

Kontener `CopyOnWriteArrayList` to specjalna implementacja interfejsu `List` do użytku w aplikacjach współbieżnych, omawiana w rozdziale „Współbieżność".

Ćwiczenie 29. Zmodyfikuj program *ListPerformance.java* tak, aby testowane kontenery zamiast egzemplarzy klasy `Integer` przechowywały obiekty klasy `String`. Do utworzenia wartości testowych wykorzystaj klasę generatora z rozdziału „Tablice" (2).

Ćwiczenie 30. Porównaj wydajność metody `Collections.sort()` dla klasy `ArrayList` i `LinkedList` (3).

Ćwiczenie 31. Utwórz kontener hermetyzujący tablicę obiektów `String`, który pozwala na dodawanie i wydobywanie jedynie obiektów `String`, więc jego użycie nie wymaga rzutowania. Jeśli rozmiar wewnętrznej tablicy kontenera nie pozwala na dodanie następnego elementu, kontener powinien automatycznie ją zwiększyć. W metodzie `main()` porównaj wydajność własnego kontenera z wydajnością kontenera `ArrayList<String>` (5).

Ćwiczenie 32. Powtórz poprzednie ćwiczenie dla kontenera wartości typu `int` i porównaj wydajność własnego kontenera z wydajnością implementacji `ArrayList<Integer>`. W porównaniu uwzględnij operację inkrementowania wszystkich elementów przechowywanych w kontenerze (2).

Ćwiczenie 33. Utwórz implementację `FastTraversalLinkedList`, która wewnętrznie bazuje na kontenerze `LinkedList` dla szybkich operacji wstawiania i usuwania, oraz kontenerze `ArrayList` służącym do szybkich operacji przeglądania i udostępniania elementów. Przetestuj swój kontener, modyfikując program *ListPerformance.java* (5).

Zagrożenia testowania w małej skali

Przy pisaniu testów w skali mikro (ang. *microbenchmarks*) trzeba zachować ostrożność oraz powstrzymać się przed zbyt daleko idącymi założeniami i zawężać testy tak, aby faktycznie mierzyć tylko to, co nas interesuje. Trzeba też zadbać o to, aby testy trwały odpowiednio długo, aby w wyniku ich realizacji doszło do wygenerowania ciekawych zbiorów danych testowych; nie bez znaczenia jest też moment załączenia technik optymalizacyjnych czasu wykonania, takich jak HotSpot — uruchamianych dopiero po pewnym czasie działania programu (trzeba to uwzględnić również przy programach krótkotrwałych).

Wyniki będą różne, zależnie od używanego komputera i zastosowanej maszyny wirtualnej, więc prezentowane testy najlepiej uruchomić samodzielnie również we własnym systemie i w ten sposób przekonać się o ich wartości. Nie warto natomiast zawracać sobie głowy bezwzględnymi liczbami podawanymi jako wyniki testów — ważniejsze są różnice pomiędzy tymi wartościami widoczne dla poszczególnych kontenerów.

W analizie wydajności sprawdza się zwykle *program profilujący*. Java posiada taki program (zobacz suplement publikowany pod adresem *http://MindView.net/Books/BetterJava*), dostępne są też tego typu programy autorstwa osób i firm trzecich, zarówno darmowe, jak i komercyjne.

Przy ilustracji zagrożeń posłużę się przykładem metody `Math.random()`. Zwraca ona wartości z zakresu od zera do jednego, ale czy włącznie z górną granicą zakresu? Czy matematycznie zbiór jej wartości to (0, 1), [0, 1], [0, 1) czy może (0, 1] (nawias prostokątny oznacza „włącznie z granicą zakresu", nawias zwykły — bez granicy zakresu)? Odpowiedzi *może* (bo nie musi) udzielić program testujący:

```
//: containers/RandomBounds.java
// Czy Math.random() zwraca 0.0 i 1.0?
// {RunByHand}
import static net.mindview.util.Print.*;

public class RandomBounds {
  static void usage() {
    print("Stosowanie:");
    print("\tRandomBounds lower");
    print("\tRandomBounds upper");
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length != 1) usage();
    if(args[0].equals("lower")) {
      while(Math.random() != 0.0)
        ; // Do skutku
      print("Produced 0.0!");
    }
    else if(args[0].equals("upper")) {
      while(Math.random() != 1.0)
        ; // Do skutku
      print("Produced 1.0!");
    }
    else
      usage();
  }
} ///:~
```

Program można uruchomić na dwa sposoby:

```
java RandomBounds lower
```

albo:

```
java RandomBounds upper
```

W obu przypadkach trzeba będzie ręcznie przerwać wykonanie programu, co pozwala sądzić, że metoda `Math.random()` nigdy nie zwraca skrajnych wartości zakresu losowania (0 i 1). Ale taki eksperyment może być mylący. Jeśli wziąć pod uwagę, że przy precyzji właściwej dla typu `double` pomiędzy zerem a jedynką można wyróżnić 262 różne wartości ułamkowe, prawdopodobieństwo wylosowania wartości skrajnych w podobnym eksperymencie może być na tyle małe, że potrzebny czas eksperymentu przekroczy czas życia komputera albo i eksperymentatora. Otóż w istocie wartość `0.0` należy do zakresu wartości metody `Math.random()`. W zapisie matematycznym wyrazilibyśmy ten zakres jako [0, 1). Widać, jak istotne jest rozumienie ograniczeń podejmowanych eksperymentów.

Wybieranie pomiędzy zbiorami

Zależnie od pożądanego zachowania można zastosować zbiór `TreeSet`, `HashSet` bądź `LinkedHashSet`. Poniższy program testowy porównuje wydajność wymienionych implementacji:

```
// : containers/SetPerformance.java
// Ilustracja wydajności implementacji interfejsu Set.
// {Args: 100 5000} Mała liczba skraca czas dla systemu kompilacji
import java.util.*;

public class SetPerformance {
  static List<Test<Set<Integer>>> tests =
    new ArrayList<Test<Set<Integer>>>();
  static {
    tests.add(new Test<Set<Integer>>("add") {
      int test(Set<Integer> set, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
          set.clear();
          for(int j = 0; j < size; j++)
            set.add(j);
        }
        return loops * size;
      }
    });
    tests.add(new Test<Set<Integer>>("contains") {
      int test(Set<Integer> set, TestParam tp) {
        int loops = tp.loops;
        int span = tp.size * 2;
        for(int i = 0; i < loops; i++)
          for(int j = 0; j < span; j++)
            set.contains(j);
        return loops * span;
      }
    });
    tests.add(new Test<Set<Integer>>("iterate") {
      int test(Set<Integer> set, TestParam tp) {
        int loops = tp.loops * 10;
        for(int i = 0; i < loops; i++) {
          Iterator<Integer> it = set.iterator();
          while(it.hasNext())
            it.next();
        }
        return loops * set.size();
```

```
      }
    });
  }
  public static void main(String[] args) {
    if(args.length > 0)
      Tester.defaultParams = TestParam.array(args);
    Tester.fieldWidth = 10;
    Tester.run(new TreeSet<Integer>(), tests);
    Tester.run(new HashSet<Integer>(), tests);
    Tester.run(new LinkedHashSet<Integer>(), tests);
  }
} /* Output: (Sample)
-------------- TreeSet --------------
rozmiar     add  contains  iterate
   10     746     173       89
  100     501     264       68
 1000     714     410       69
10000    1975     552       69
-------------- HashSet --------------
rozmiar     add  contains  iterate
   10     308      91       94
  100     178      75       73
 1000     216     110       72
10000     711     215      100
------------ LinkedHashSet ------------
rozmiar     add  contains  iterate
   10     350      65       83
  100     270      74       55
 1000     303     111       54
10000    1615     256       58
*///:~
```

Zasadniczo HashSet pod względem wydajności wygrywa z implementacją TreeSet; różnicę widać najbardziej przy dodawaniu elementów i ich wyszukiwaniu, czyli dwóch najważniejszych operacjach. Podstawową cechą TreeSet jest przechowywanie elementów z ich porządkowaniem, więc kontener ten należy wykorzystywać głównie tam, gdzie potrzebne są posortowane zbiory (Set) elementów. Struktura wewnętrzna predestynuje tę implementację do obsługi sortowania i ze względu na powszechność operacji przeglądania zbioru (iteracji) przeglądanie jest zwykle szybsze dla TreeSet niż dla HashSet.

Wstawianie do kontenera LinkedHashSet jest operacją kosztowniejszą niż analogiczna operacja na kontenerze HashSet, a to z powodu dodatkowego narzutu wynikającego z utrzymywania wewnętrznej listy elementów reprezentującej ich pierwotną kolejność.

Ćwiczenie 34. Zmodyfikuj program *SetPerformance.java* tak, aby zbiory przechowywały nie wartości Integer, a obiekty typu String. Do wygenerowania ciągów testowych wykorzystaj klasę Generator z rozdziału „Tablice" (1).

Wybieranie pomiędzy odwzorowaniami

Poniższy program porównuje wydajność implementacji interfejsu Map:

```
//: containers/MapPerformance.java
// Ilustracja wydajności różnych implementacji Map.
// {Args: 100 5000} Mała liczba skraca czas dla systemu kompilacji
import java.util.*;
```

```
public class MapPerformance {
  static List<Test<Map<Integer,Integer>>> tests =
    new ArrayList<Test<Map<Integer,Integer>>>();
  static {
    tests.add(new Test<Map<Integer,Integer>>("put") {
      int test(Map<Integer,Integer> map, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
          map.clear();
          for(int j = 0; j < size; j++)
            map.put(j, j);
        }
        return loops * size;
      }
    });
    tests.add(new Test<Map<Integer,Integer>>("get") {
      int test(Map<Integer,Integer> map, TestParam tp) {
        int loops = tp.loops;
        int span = tp.size * 2;
        for(int i = 0; i < loops; i++)
          for(int j = 0; j < span; j++)
            map.get(j);
        return loops * span;
      }
    });
    tests.add(new Test<Map<Integer,Integer>>("iterate") {
      int test(Map<Integer,Integer> map, TestParam tp) {
        int loops = tp.loops * 10;
        for(int i = 0; i < loops; i ++) {
          Iterator it = map.entrySet().iterator();
          while(it.hasNext())
            it.next();
        }
        return loops * map.size();
      }
    });
  }
  public static void main(String[] args) {
    if(args.length > 0)
      Tester.defaultParams = TestParam.array(args);
    Tester.run(new TreeMap<Integer,Integer>(), tests);
    Tester.run(new HashMap<Integer,Integer>(), tests);
    Tester.run(new LinkedHashMap<Integer,Integer>(),tests);
    Tester.run(
      new IdentityHashMap<Integer,Integer>(), tests);
    Tester.run(new WeakHashMap<Integer,Integer>(), tests);
    Tester.run(new Hashtable<Integer,Integer>(), tests);
  }
} /* Output: (Sample)
------------ TreeMap ------------
 rozmiar    put    get iterate
    10    748    168    100
   100    506    264     76
  1000    771    450     78
 10000   2962    561     83
------------ HashMap ------------
```

```
rozmiar    put    get iterate
     10    281     76     93
    100    179     70     73
   1000    267    102     72
  10000   1305    265     97
--------- LinkedHashMap ---------
rozmiar    put    get iterate
     10    354    100     72
    100    273     89     50
   1000    385    222     56
  10000   2787    341     56
-------- IdentityHashMap --------
rozmiar    put    get iterate
     10    290    144    101
    100    204    287    132
   1000    508    336     77
  10000    767    266     56
---------- WeakHashMap ----------
rozmiar    put    get iterate
     10    484    146    151
    100    292    126    117
   1000    411    136    152
  10000   2165    138    555
---------- Hashtable ----------
rozmiar    put    get iterate
     10    264    113    113
    100    181    105     76
   1000    260    201     80
  10000   1245    134     77
*///:~
```

Operacje wstawiania elementów do wszystkich kontenerów asocjacyjnych z wyjątkiem `IdentityHashMap` znacząco wydłużają się w miarę wzrostu rozmiaru kontenera. Zasadniczo jednak wyszukiwanie jest znacznie efektywniejsze od wstawiania, co jest o tyle korzystne, że w takich kontenerach częściej szuka się elementów niż je do nich wstawia.

Wydajność `Hashtable` jest zbliżona do wydajności `HashMap`. Ponieważ docelowo `HashMap` ma wyprzeć kontener `Hashtable` z użycia, nie powinno dziwić, że oba wykorzystują identyczny mechanizm wewnętrznego przechowywania i identyczny mechanizm wyszukiwania (o którym później).

Implementacja `TreeMap` jest w przypadku ogólnym wolniejsza od `HashMap`. Tak jak `TreeSet`, `TreeMap` służy do tworzenia zbiorów uporządkowanych. Oparcie implementacji na drzewie sprawia, że elementy wstawiane do kontenera są zawsze uporządkowane względem siebie. Po wypełnieniu kontenera `TreeMap` można za pomocą metody `keySet()` pozyskać zbiór-perspektywę kluczy odwzorowania, a następnie zamienić go na tablicę metodą `toArray()`. Następnie można skorzystać ze statycznej metody `Arrays.binarySearch()` w celu szybkiego wyszukiwania obiektów w posortowanej tablicy. Oczywiście stosuje się to tylko tam, gdzie niepożądane jest zachowanie właściwe implementacji `HashMap` — bo normalnie dużą szybkość wyszukiwania kluczy osiąga się właśnie za pomocą kontenerów `HashMap`. Do tego w każdej chwili można na podstawie `TreeSet` utworzyć kontener `HashMap`, za pomocą prostej konstrukcji czy wywołania `putAll()`. Tak czy owak przy wyborze kontenera asocjacyjnego pierwszą kandydaturą powinna być zawsze implementacja `HashMap`; `TreeMap` należy wybierać tylko tam, gdzie potrzebne jest uporządkowanie elementów odwzorowania według wartości.

Implementacja LinkedHashMap jest zazwyczaj wolniejsza od HashMap w operacjach wstawiania, bo utrzymuje dodatkową listę elementów reprezentującą ich pierwotną kolejność. Za to ta dodatkowa lista zapewnia szybsze przeglądanie kontenera.

Kontenery IdentityHashMap cechują się inną wydajnością, bo przy porównywaniu zamiast wywołań equals() stosują operator ==. Kontenerem WeakHashMap zajmiemy się osobno.

Ćwiczenie 35. Zmodyfikuj program *MapPerformance.java* tak, aby wykorzystać w nim kontener SlowMap (1).

Ćwiczenie 36. Zmień klasę SlowMap tak, aby zamiast dwóch kontenerów ArrayList wykorzystywała pojedynczą listę (ArrayList) obiektów MapEntry. Sprawdź, czy zmodyfikowana wersja działa poprawnie. Za pomocą programu *MapPerformance.java* przetestuj szybkość działania nowej implementacji. Potem zmień metodę put() tak, aby po wstawieniu każdej kolejnej pary sortowała zawartość kontenera i metodę get() tak, aby szukała klucza wywołaniem Collections.binarySearch(). Porównaj wydajność obu implementacji z implementacjami pierwotnymi (5).

Ćwiczenie 37. Zmień klasę SimpleHashMap tak, aby zamiast kontenerów LinkedList wykorzystywała kontenery ArrayList. Zmodyfikuj program *MapPerformance.java* pod kątem testu obu implementacji (nowej i pierwotnej) (2).

Czynniki wydajności HashMap

Kontener HashMap można ręcznie „podkręcać" w celu dalszego zwiększenia jego wydajności w danej aplikacji. Aby ogarnąć kwestie wydajnościowe związane z takim dostrajaniem, należałoby ustalić podstawową terminologię:

Czynnik	Znaczenie
Pojemność	Liczba komórek tablicy.
Pojemność początkowa	Liczba komórek tablicy tuż po jej stworzeniu. Klasa HashMap i HashSet posiadają konstruktory pozwalające na określenie pojemności początkowej.
Rozmiar	Liczba pozycji znajdujących się w danej chwili w tablicy.
Współczynnik wypełnienia	Rozmiar/pojemność. Współczynnik wypełnienia o wartości zero oznacza, że tablica jest pusta, 0,5 — w połowie wypełniona itd. Mało obciążona tablica będzie miała kilka kolizji i będzie optymalna do wstawiania i wyszukiwania (ale spowolni proces dokładnego sprawdzania z użyciem iteratora). HashMap i HashSet mają konstruktory pozwalające określić współczynnik wypełnienia, co oznacza, że gdy współczynnik ten zostanie osiągnięty, to kontener automatycznie zwiększy swoją pojemność (liczbę kubełków) poprzez podwojenie, a następnie ponownie przydzieli przechowywane obiekty do nowych kubełków (nazywa się to powtórnym haszowaniem).

Domyślny współczynnik wypełnienia stosowany przez HashMap wynosi 0,75 (nie wykonuje powtórnego haszowania, dopóki tablica nie jest w 3/4 zajęta). Wydaje się to dobrym kompromisem pomiędzy kosztami czasowymi a potrzebną pamięcią. Wyższy współczynnik wypełnienia zmniejsza ilość miejsca wymaganą przez tablicę, ale zwiększa czas wyszukiwania, co jest ważne, ponieważ najczęściej wykonuje się właśnie przeszukiwanie (zarówno w get(), jak i w put()).

Jeżeli wiemy, że musimy przechowywać dużo elementów w HashMap, to stworzenie go z odpowiednio dużą pojemnością początkową uchroni nas przed kosztami ponownego haszowania[12].

Ćwiczenie 38. Odszukaj w dokumentacji JDK opis klasy HashMap. Stwórz obiekt tej klasy, wypełnij go elementami i wyznacz współczynnik wypełnienia. Zbadaj szybkość wyszukiwania w tym odwzorowaniu, a potem spróbuj ją zwiększyć poprzez stworzenie nowego obiektu HashMap z większą pojemnością początkową i skopiowanie wcześniejszego odwzorowania do tego nowego. Uruchom test sprawdzania prędkości na nowym odwzorowaniu (3).

Ćwiczenie 39. Dodaj do SimpleHashMap prywatną metodę rehaszującą rehash(), która będzie wywoływana, kiedy współczynnik obciążenia przekroczy wartość 0,75. Podczas ponownego haszowania zwiększ dwukrotnie liczbę kubełków, a następnie wyszukaj pierwszą liczbę pierwszą większą niż owe podwojenie; ta liczba pierwsza ma określać nową liczbę kubełków (6).

Narzędzia dodatkowe

Kontenerom towarzyszy zestaw narzędzi pomocniczych w postaci statycznych metod klasy java.util.Collections. Znasz już niektóre z nich, jak choćby addAll(), reverseOrder() czy binarySearch(). Oto pozostałe (narzędzia synchronizowania i „niemodyfikowalności" zostaną omówione w następnych podrozdziałach); tam, gdzie to ważne, tabela wymienia uogólnienia występujące w omawianych narzędziach:

Metoda	Działanie
checkedCollection(Collection<T>, Class<T> typ) checkedList(List<T>, Class<T> typ) checkedMap(Map<K,V>, Class<K> tKlucza, Class<V> tWartości) checkedSet(Set<T>, Class<T> typ) checkedSortedMap(SortedMap<K,V>, Class<K> tKlucza, Class<V> tWartości) checkedSortedSet(SortedSet<T>, Class<T> typ)	Tworzą *dynamicznie kontrolowaną* (odnośnie typu) perspektywę kolekcji Collection albo jej konkretnego podtypu. Do użycia tam, gdzie nie można zastosować wersji kontrolowanych statycznie. Omawiane w rozdziale „Typy ogólne" w podrozdziale „Dynamiczna kontrola typów".

[12] W prywatnej korespondencji Joshua Bloch napisał: „...Wierzę, że popełniamy błąd, udostępniając w naszych API możliwości zmiany szczegółów implementacji (takich jak wielkość tablicy haszującej oraz współczynnik wypełnienia). Być może klient powinien podawać maksymalną dopuszczalną wielkość kolekcji, a wszelkie pozostałe informacje powinny być obliczane na jej podstawie. Podając wartości innych parametrów, klienci mogą wyrządzić więcej szkody niż pożytku. Ekstremalnym przykładem jest składowa capacityIncrement klasy Vector. Nikt nigdy nie powinien zmieniać pojemności wektora, a my nie powinniśmy udostępniać tej metody. W przypadku przypisania jej dowolnej wartości różnej od zera koszt sekwencji operacji dodawania elementów do wektora zmienia się z liniowego na kwadratowy. Innymi słowy, efektywność działania klasy jest całkowicie tracona. Jeśli chodzi o takie rozwiązania, to zaczynamy mądrzeć wraz z upływem czasu. Jeśliby przeanalizować klasę IdentityHashMap, okaże się, że nie ma w niej żadnych narzędzi niskiego poziomu służących do regulacji jej działania."

Metoda	Działanie
`max(Collection)` `min(Collection)`	Zwraca maksymalny i minimalny element spośród zamieszczonych w kontenerze-argumencie, stosując przy tym normalną metodę porównania dla obiektów zawartych w strukturze.
`max(Collection, Comparator)` `min(Collection, Comparator)`	Zwraca maksymalny i minimalny element kontenera, stosując wskazany przez `Comparator` sposób porównania.
`indexOfSubList` `(List źródło, List cel)`	Określa początkowy indeks pierwszego miejsca, w którym lista `cel` występuje w liście `źródło`.
`lastIndexOfSubList` `(List źródło, List cel)`	Określa początkowy indeks ostatniego miejsca, w którym lista `cel` występuje w liście `źródło`.
`replaceAll(List<T> lista, T staraWart, T nowaWart)`	Zastępuje wszystkie obiekty `staraWart` obiektami `nowaWart`.
`reverse(List)`	Odwracanie kolejności występowania elementów w przekazanej liście.
`reverseOrder()` `reverseOrder(Comparator<T>)`	Zwraca obiekt `Comparator` odwracający naturalny porządek obiektów w kolekcji. Druga wersja odwraca porządek wyznaczany przez przekazany komparator.
`rotate(List lista, int dystans)`	Przesuwa wszystkie elementy listy o określony dystans, pobierając elementy z końca listy i umieszczając je na jej początku.
`shuffle(List)` `shuffle(List, Random)`	Permutuje (losowo) wskazaną listę. Pierwsza wersja korzysta z własnego generatora losowego, można go jednak zastąpić własnym (w drugiej wersji metody).
`sort(List<T>)` `sort(List<T>, Comparator<? super T> c)`	Sortuje listę `List<T>` według naturalnego porządku elementów. Druga wersja pozwala na zastosowanie własnego komparatora ustalającego kolejność elementów.
`copy(List<? super T> cel, List<? extends T> źródło)`	Kopiowanie elementów ze źródła do celu.
`swap(List lista, int i, int j)`	Zamienia położenie elementów `i` i `j` na liście `lista`.
`fill(List<? super T> lista, T x)`	Zastąpienie wszystkich obiektów listy przez obiekt `x`.
`nCopies(int n, T x)`	Zwraca niemodyfikowalną listę `List<T>` rozmiaru `n`, której wszystkie odwołania wskazują na obiekt `x`.
`disjoint(Collection, Collection)`	Zwraca `true`, jeśli w obu kolekcjach nie występują wspólne elementy.
`frequency(Collection, Object x)`	Zwraca liczbę elementów kolekcji `Collection` równych elementowi `x`.

`emptyList()`
`emptyMap()`
`emptySet()`
	Zwraca niemodyfikowalną listę (`List`), zbiór (`Set`) albo odwzorowanie (`Map`). To uogólnienia, więc wynikowa kolekcja będzie parametryzowana pożądanym typem.

`singleton(T x)`
`singletonList(T x)`
`singletonMap(K klucz, V wartość)`
	Zwraca niemodyfikowalny zbiór (`Set<T>`), listę (`List<T>`) bądź odwzorowanie (`Map<K,V>`) z pojedynczym elementem wybranym na podstawie argumentu `x`.

`list(Enumeration<T> e)`	Zwraca obiekt `ArrayList<T>` wygenerowany przy użyciu podanego obiektu `Enumeration`. Służy do aktualizacji kodu tworzonego dla wcześniejszych wersji JDK.
`enumeration(Collection<T>)`	Zwraca obiekt `Enumeration<T>` dla podanego argumentu.

Zauważ, iż metody `min()` i `max()` działają na obiektach `Collection`, a nie `List`, dlatego nie ma potrzeby martwić się o to, czy kolekcja powinna być posortowana czy nie (jak wspominałem wcześniej, *trzeba* posortować listę `List` lub tablicę przed wykonaniem `binarySearch()`).

Oto przykład pokazujący najprostsze zastosowania większości narzędzi wymienionych w powyższej tabeli:

```
//: containers/Utilities.java
// Demonstracja oprzyrządowania z klasy Collections.
import java.util.*;
import static net.mindview.util.Print.*;

public class Utilities {
  static List<String> list = Arrays.asList(
    "jeden Dwa trzy Cztery pięć sześć jeden".split(" "));
  public static void main(String[] args) {
    print(list);
    print("Rozłączność list (Cztery)?: " +
      Collections.disjoint(list,
        Collections.singletonList("Cztery")));
    print("max: " + Collections.max(list));
    print("min: " + Collections.min(list));
    print("max z komparatorem: " + Collections.max(list,
      String.CASE_INSENSITIVE_ORDER));
    print("min z komparatorem: " + Collections.min(list,
      String.CASE_INSENSITIVE_ORDER));
    List<String> sublist =
      Arrays.asList("Cztery pięć sześć".split(" "));
    print("indexOfSubList: " +
      Collections.indexOfSubList(list, sublist));
    print("lastIndexOfSubList: " +
      Collections.lastIndexOfSubList(list, sublist));
    Collections.replaceAll(list, "jeden", "Uj");
    print("replaceAll: " + list);
    Collections.reverse(list);
    print("reverse: " + list);
    Collections.rotate(list, 3);
    print("rotate: " + list);
    List<String> source =
      Arrays.asList("we wnętrzu macierzy".split(" "));
    Collections.copy(list, source);
    print("copy: " + list);
    Collections.swap(list, 0, list.size() - 1);
    print("swap: " + list);
    Collections.shuffle(list, new Random(47));
    print("shuffle: " + list);
```

```
    Collections.fill(list, "pop");
    print("fill: " + list);
    print("częstotliwość słowa 'pop': " +
      Collections.frequency(list, "pop"));
    List<String> dups = Collections.nCopies(3, "pstryk");
    print("duplikaty: " + dups);
    print("rozłączność list ('duplikaty')?: " +
      Collections.disjoint(list, dups));
    // Pozyskiwanie enumeratora:
    Enumeration<String> e = Collections.enumeration(dups);
    Vector<String> v = new Vector<String>();
    while(e.hasMoreElements())
      v.addElement(e.nextElement());
    // Konwersja kontenera Vector
    // na listę List za pomocą enumeratora:
    ArrayList<String> arrayList =
      Collections.list(v.elements());
    print("arrayList: " + arrayList);
  }
} /* Output:
[jeden, Dwa, trzy, Cztery, pięć, sześć, jeden]
Rozłączność list (Cztery)?: false
max: trzy
min: Cztery
max z komparatorem: trzy
min z komparatorem: Cztery
indexOfSubList: 3
lastIndexOfSubList: 3
replaceAll: [Uj, Dwa, trzy, Cztery, pięć, sześć, Uj]
reverse: [Uj, sześć, pięć, Cztery, trzy, Dwa, Uj]
rotate: [trzy, Dwa, Uj, Uj, sześć, pięć, Cztery]
copy: [we, wnętrzu, macierzy, Uj, sześć, pięć, Cztery]
swap: [Cztery, wnętrzu, macierzy, Uj, sześć, pięć, we]
shuffle: [sześć, macierzy, wnętrzu, Cztery, Uj, pięć, we]
fill: [pop, pop, pop, pop, pop, pop, pop]
częstotliwość słowa 'pop': 7
duplikaty: [pstryk, pstryk, pstryk]
rozłączność list ('duplikaty')?: true
arrayList: [pstryk, pstryk, pstryk]
*///:~
```

Generowane wyniki wyjaśniają działanie poszczególnych metod pomocniczych. Warto zwrócić uwagę na różnice w działaniu metody `min()` oraz `max()` wykorzystujących komparator `String.CASE_INSENSITIVE_ORDER`, wynikające z uwzględnienia wielkości liter.

Sortowanie i przeszukiwanie list

Narzędzia do sortowania i przeszukiwania kontenerów `List` posiadają nazwy i sygnatury identyczne z tymi przeznaczonymi dla tablic obiektów, tyle że owe narzędzia są statycznymi metodami klasy `Collections`, a nie `Arrays`. Oto przykład wykorzystujący listę danych z przykładu *Utilities.java*:

```
//: containers/ListSortSearch.java
// Sortowanie i wyszukiwanie w listach
// za pomocą oprzyrządowania Collections.
import java.util.*;
import static net.mindview.util.Print.*;
```

```
public class ListSortSearch {
  public static void main(String[] args) {
    List<String> list =
      new ArrayList<String>(Utilities.list);
    list.addAll(Utilities.list);
    print(list);
    Collections.shuffle(list, new Random(47));
    print("Potasowana: " + list);
    // Zastosowanie iteratora ListIterator do obcięcia ostatniego elementu:
    ListIterator<String> it = list.listIterator(10);
    while(it.hasNext()) {
      it.next();
      it.remove();
    }
    print("Przycięta: " + list);
    Collections.sort(list);
    print("Posortowana: " + list);
    String key = list.get(7);
    int index = Collections.binarySearch(list, key);
    print("Pozycja " + key + " to " + index +
      ", list.get(" + index + ") = " + list.get(index));
    Collections.sort(list, String.CASE_INSENSITIVE_ORDER);
    print("Posortowana (bez rozpoznawania wielkości znaków): " + list);
    key = list.get(7);
    index = Collections.binarySearch(list, key,
      String.CASE_INSENSITIVE_ORDER);
    print("Pozycja " + key + " to " + index +
      ", list.get(" + index + ") = " + list.get(index));
  }
} /* Output:
[jeden, Dwa, trzy, Cztery, pięć, sześć, jeden, jeden, Dwa, trzy, Cztery, pięć, sześć, jeden]
Potasowana: [Cztery, pięć, jeden, jeden, Dwa, sześć, sześć, trzy, trzy, pięć, Cztery, Dwa, jeden, jeden]
Przycięta: [Cztery, pięć, jeden, jeden, Dwa, sześć, sześć, trzy, trzy, pięć]
Posortowana: [Cztery, Dwa, jeden, jeden, pięć, pięć, sześć, sześć, trzy, trzy]
Pozycja sześć to 7, list.get(7) = sześć
Posortowana (bez rozpoznawania wielkości znaków): [Cztery, Dwa, jeden, jeden, pięć, pięć, sześć, sześć,
trzy, trzy]
Pozycja sześć to 7, list.get(7) = sześć
*///:~
```

Jak przy przeszukiwaniu tablic, jeśli sortowanie odbywało się na podstawie pewnego komparatora, wyszukiwanie binarne (`binarySearch()`) powinno również wykorzystywać ten sam komparator.

Powyższy program pokazuje przy okazji działanie metody `shuffle()` klasy `Collections`. Służy ona do tasowania elementów listy, tak aby zostały przemieszczone względem siebie na losowe pozycje. Tworzony potem iterator `ListIterator` wskazuje wybraną pozycję potasowanej listy i służy do usunięcia elementów od tej pozycji do końca listy.

Ćwiczenie 40. Utwórz klasę zawierającą parę obiektów `String` i zaimplementuj dla niej interfejs `Comparable` tak, aby w porównaniach uwzględniane były jedynie pierwsze ciągi `String`. Za pomocą generatora `RandomGenerator` wypełnij tablicę i kontener `ArrayList` obiektami tej klasy. Wykaż, że sortowanie takich kolekcji działa poprawnie. Teraz zdefiniuj komparator `Comparator`, który będzie porównywał egzemplarze klasy na bazie drugiego obiektu `String` i pokaż, że i teraz sortowanie działa poprawnie. Wykorzystaj też komparator do realizacji wyszukiwania binarnego (5).

Ćwiczenie 41. Zmodyfikuj klasę z poprzedniego ćwiczenia tak, aby jej obiekty nadawały się do przechowywania w kontenerach HashSet i mogły pełnić rolę kluczy w kontenerach HashMap (3).

Ćwiczenie 42. Zmodyfikuj ćwiczenie 40. tak, aby sortowanie było alfabetyczne (2).

Niemodyfikowalne kontenery Collection i Map

Często wygodne jest stworzenie kolekcji lub odwzorowania w wersji tylko do odczytu. Klasa Collections pozwala na wykonanie tego poprzez przekazanie oryginalnego kontenera do metody, która zwraca wersję tylko do odczytu. Istnieje kilka odmian wspomnianej metody, po jednej specjalnie dla Collection (jeśli nie możemy traktować kontenera jako ściślej określonego), List, Set i Map. Zamieszczony przykład pokazuje właściwy sposób budowy wersji tylko do odczytu każdego z tych rodzajów.

```
//: containers/ReadOnly.java
// Zastosowanie metod Collections.unmodifiable...
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ReadOnly {
  static Collection<String> data =
    new ArrayList<String>(Countries.names(6));
  public static void main(String[] args) {
    Collection<String> c =
      Collections.unmodifiableCollection(
        new ArrayList<String>(data));
    print(c); // Odczyt jest dozwolony
    //! c.add("jeden"); // Ale modyfikacje już nie

    List<String> a = Collections.unmodifiableList(
        new ArrayList<String>(data));
    ListIterator<String> lit = a.listIterator();
    print(lit.next()); // Odczyt jest dozwolony
    //! lit.add("jeden"); // Ale modyfikacje już nie

    Set<String> s = Collections.unmodifiableSet(
      new HashSet<String>(data));
    print(s); // Odczyt jest dozwolony
    //! s.add("jeden"); // Ale modyfikacje już nie

    // Dla kontenera SortedSet:
    Set<String> ss = Collections.unmodifiableSortedSet(
      new TreeSet<String>(data));

    Map<String,String> m = Collections.unmodifiableMap(
      new HashMap<String,String>(Countries.capitals(6)));
    print(m); // Odczyt jest dozwolony
    //! m.put("Romek", "Hejka!");

    // Dla kontenera SortedMap:
    Map<String,String> sm =
      Collections.unmodifiableSortedMap(
        new TreeMap<String,String>(Countries.capitals(6)));
  }
```

```
} /* Output:
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
ALGERIA
[BULGARIA, BURKINA FASO, BOTSWANA, BENIN, ANGOLA, ALGERIA]
{BULGARIA=Sofia, BURKINA FASO=Ouagadougou, BOTSWANA=Gaberone, BENIN=Porto-Novo,
ANGOLA=Luanda, ALGERIA=Algiers}
*///:~
```

Wywołanie metody zapewniającej niemodyfikowalność dla konkretnego typu nie uaktywnia sprawdzania w czasie kompilacji, ale dopiero w trakcie wykonania, zaraz po wystąpieniu transformacji, jakiekolwiek wywołanie metody modyfikującej zawartość kontenera spowoduje zgłoszenie wyjątku `UnsupportedOperationException`.

W każdym przypadku trzeba wypełnić kontener potrzebnymi danymi, zanim uczyni się go niemodyfikowalnym. Po zapełnieniu najlepszym rozwiązaniem jest zastąpienie istniejącej referencji referencją uzyskaną z wywołania metody typu „unmodifiable". Tym sposobem pozbywamy się ryzyka przypadkowej zmiany zawartości już po zamianie w kontener niemodyfikowalny. Z drugiej strony, narzędzie to pozwala także na zatrzymanie modyfikowalnej wersji kontenera jako prywatnego w klasie i zwrot referencji tylko do odczytu poprzez wywołanie jakiejś metody. Tak więc możemy go zmieniać wewnątrz klasy, ale ktoś inny może wyłącznie odczytać zawartość.

Synchronizacja Collection i Map

Słowo kluczowe `synchronized` stanowi istotną część tematu *wielowątkowości* — bardzo skomplikowanego zagadnienia, który będzie poruszone dopiero w rozdziale „Współbieżność". Tutaj jedynie zaznaczę, że klasa `Collections` udostępnia sposób automatycznej synchronizacji całego kontenera. Składnia jest podobna do metod tworzących niemodyfikowalne wersje kontenerów:

```
//: containers/Synchronization.java
// Zastosowanie metod Collections.synchronized.
import java.util.*;

public class Synchronization {
  public static void main(String[] args) {
    Collection<String> c =
      Collections.synchronizedCollection(
        new ArrayList<String>());
    List<String> list = Collections.synchronizedList(
      new ArrayList<String>());
    Set<String> s = Collections.synchronizedSet(
      new HashSet<String>());
    Set<String> ss = Collections.synchronizedSortedSet(
      new TreeSet<String>());
    Map<String,String> m = Collections.synchronizedMap(
      new HashMap<String,String>());
    Map<String,String> sm =
      Collections.synchronizedSortedMap(
        new TreeMap<String,String>());
  }
} ///:~
```

Najlepiej, jak w tym przypadku, natychmiast przekazać nowy kontener do odpowiedniej metody synchronizującej; dzięki temu nie ma sposobności przypadkowego udostępnienia wersji niesynchronizowanej.

Mechanizm fail-fast

Kontenery Javy posiadają również mechanizm zabezpieczający przed sytuacją, w której więcej niż jeden proces modyfikuje zawartość kontenera. Problem pojawia się, jeśli przeglądamy kontener i wkroczy kilka innych procesów, by wstawić, usunąć lub zmienić jakiś obiekt zamieszczony w kontenerze. Może obiekt został już ominięty, a może dopiero to nastąpi, może rozmiar kontenera zmniejszy się po wywołaniu `size()` — istnieje wiele scenariuszy prowadzących do katastrofy. Biblioteka kontenerów w Javie uwzględnia mechanizm *fail-fast*, który wyszukuje wszystkie zmiany kontenera inne niż dokonane w naszym procesie. Jeżeli odkryje, że ktoś inny modyfikuje kontener, od razu spowoduje to pojawienie się wyjątku `ConcurrentModificationException`. Oto zaleta *fail-fast* — nie próbuje wykryć problemu później, stosując bardziej skomplikowane algorytmy.

Obejrzenie tego mechanizmu w działaniu jest stosunkowo proste — wszystko, co trzeba zrobić, to stworzyć iterator, a następnie dodać coś do kolekcji, na którą iterator wskazuje:

```
//: containers/FailFast.java
// Demonstracja mechanizmu "fail-fast".
import java.util.*;

public class FailFast {
  public static void main(String[] args) {
    Collection<String> c = new ArrayList<String>();
    Iterator<String> it = c.iterator();
    c.add("Obiekt");
    try {
      String s = it.next();
    } catch(ConcurrentModificationException e) {
      System.out.println(e);
    }
  }
} /* Output:
java.util.ConcurrentModificationException
*///:~
```

Wyjątek zostanie zgłoszony, ponieważ dokładamy coś do kontenera po uzyskaniu z niego iteratora. Możliwość, że dwie części programu mogą modyfikować ten sam kontener, powoduje stan niepewny, dlatego wyjątek informuje, iż powinniśmy zmienić kod — w tym przypadku pobrać iterator *po* dodaniu wszystkich elementów do kontenera.

W klasach `ConcurrentHashMap`, `CopyOnWriteArrayList` i `CopyOnWriteArraySet` stosowane są techniki unikania wyjątków `ConcurrentModificationException`.

Przechowywanie referencji

Biblioteka `java.lang.ref` zawiera zestaw klas pozwalających na lepszą współpracę z odśmiecaczem pamięci. Mogą być one szczególnie użyteczne w przypadku dużych obiektów, mogących powodować wyczerpanie pamięci. Mamy tu trzy klasy dziedziczące po `Reference`:

SoftReference, WeakReference i PhantomReference. Każda z nich zapewnia inny poziom odśmiecania, jeżeli rozpatrywany obiekt jest dostępny *tylko* przez jeden z tych obiektów klasy Reference.

Jeśli jakiś obiekt jest *osiągalny*, oznacza to, że w jakiś sposób w programie obiekt ten można odnaleźć. Może to znaczyć, że mamy na stosie zwykłą referencję wskazującą obiekt, ale również możemy mieć referencję do obiektu zawierającego referencję do obiektu rozważanego; może być wiele takich pośrednich odwołań. Jeżeli obiekt jest osiągalny, to odśmiecacz nie może go zwolnić, gdyż wciąż może być używany w programie. Jeżeli natomiast obiekt jest niedostępny — nie ma możliwości jego użycia, to jego uprzątnięcie jest bezpieczne.

Obiektów Reference używa się, kiedy nadal chce się przechowywać referencję do obiektu — aby można było sięgnąć do tego obiektu — ale również aby pozwolić odśmiecaczowi pamięci na zwolnienie takiego obiektu. Zatem mamy sposób na dalsze używanie obiektu, ale jeśli dostępna pamięć jest bliska wyczerpania, to zezwalamy na zwolnienie obiektu.

Możemy to osiągnąć przez zastosowanie obiektu Reference jako pośrednika pomiędzy nami a zwykłą referencją, *przy czym* nie może istnieć żadna zwykła referencja do obiektu (nie schowana wewnątrz obiektu klasy Reference). Gdy odśmiecacz pamięci odkryje, że obiekt jest osiągalny poprzez zwykłą referencję, nie zniszczy go.

Klasa SoftReference, WeakReference i PhantomReference — w tej kolejności — są coraz „słabsze" i odpowiadają różnym poziomom dostępności. SoftReference służy do implementacji pamięci podręcznych. WeakReference jest przeznaczona do implementacji „odwzorowań kanonicznych" — w których egzemplarze obiektów mogą być równocześnie używane w wielu miejscach programu, aby oszczędzić pamięć — nie odbierają możliwości swym kluczom (lub wartościom) na odśmiecanie. Natomiast klasa PhantomReference służy do wykonywania akcji związanych ze sprzątaniem obiektu tuż przed zniszczeniem w elastyczniejszy sposób niż poprzez mechanizm finalizacji w Javie.

W przypadku obiektów SoftReference i WeakReference można wybrać, czy zamieścić je w kolejce ReferenceQueue (przechowującej obiekty wymagające wykonania sprzątania przed śmiercią), ale PhantomReference może być zbudowana tylko w oparciu o ReferenceQueue. Oto prosty przykład:

```
//: containers/References.java
// Demonstracja zastosowania obiektów Reference
import java.lang.ref.*;
import java.util.*;

class VeryBig {
  private static final int SIZE = 10000;
  private long[] la = new long[SIZE];
  private String ident;
  public VeryBig(String id) { ident = id; }
  public String toString() { return ident; }
  protected void finalize() {
    System.out.println("Finalizacja " + ident);
  }
}
```

```
public class References {
  private static ReferenceQueue<VeryBig> rq =
    new ReferenceQueue<VeryBig>();
  public static void checkQueue() {
    Reference<? extends VeryBig> inq = rq.poll();
    if(inq != null)
      System.out.println("W kolejce: " + inq.get());
  }
  public static void main(String[] args) {
    int size = 10;
    // Albo wybór rozmiaru z wiersza poleceń:
    if(args.length > 0)
      size = new Integer(args[0]);
    LinkedList<SoftReference<VeryBig>> sa =
      new LinkedList<SoftReference<VeryBig>>();
    for(int i = 0; i < size; i++) {
      sa.add(new SoftReference<VeryBig>(
        new VeryBig("Soft " + i), rq));
      System.out.println("Właśnie utworzony: " + sa.getLast());
      checkQueue();
    }
    LinkedList<WeakReference<VeryBig>> wa =
      new LinkedList<WeakReference<VeryBig>>();
    for(int i = 0; i < size; i++) {
      wa.add(new WeakReference<VeryBig>(
        new VeryBig("Weak " + i), rq));
      System.out.println("Właśnie utworzony: " + wa.getLast());
      checkQueue();
    }
    SoftReference<VeryBig> s =
      new SoftReference<VeryBig>(new VeryBig("Soft"));
    WeakReference<VeryBig> w =
      new WeakReference<VeryBig>(new VeryBig("Weak"));
    System.gc();
    LinkedList<PhantomReference<VeryBig>> pa =
      new LinkedList<PhantomReference<VeryBig>>();
    for(int i = 0; i < size; i++) {
      pa.add(new PhantomReference<VeryBig>(
        new VeryBig("Phantom " + i), rq));
      System.out.println("Właśnie utworzony: " + pa.getLast());
      checkQueue();
    }
  }
} /* (Execute to see output) *///:~
```

Po uruchomieniu programu (wyjście najlepiej przekierować do pliku, aby można je było spokojnie przeanalizować) zobaczysz, że obiekty są niszczone przez odśmiccacz nawet wtedy, gdy wciąż mamy do nich dostęp poprzez obiekt Reference (by uzyskać rzeczywiste odwołanie do obiektu, stosuje się metodę get()). Zauważysz również, że ReferenceQueue zawsze zwraca Reference zawierającą obiekt null. Aby to stosować, można wydziedziczyć nową klasę z odpowiedniej klasy Reference i dodać jej więcej użytecznych metod.

WeakHashMap

Biblioteka kontenerów zawiera specjalne odwzorowanie Map do przechowywania słabych referencji — WeakHashMap. Klasa ta jest zaprojektowana w celu ułatwienia tworzenia odwzorowań kanonicznych. W takim odwzorowaniu oszczędzamy pamięć poprzez stworzenie tylko jednego egzemplarza danej wartości. Kiedy program potrzebuje takiej wartości, szuka istniejącego obiektu w odwzorowaniu i właśnie go wykorzystuje (zamiast od podstaw tworzyć nowy). Odwzorowanie może stworzyć wartości w czasie swej inicjalizacji, ale bardziej prawdopodobne jest, że będą one tworzone na żądanie.

Ponieważ jest to technika oszczędzania pamięci, WeakHashMap pozwala odśmiecaczowi na automatyczne czyszczenie kluczy i wartości. Nie musimy robić niczego szczególnego wobec kluczy i wartości, które chcielibyśmy zamieścić w obiekcie klasy WeakHashMap; są one automatycznie opakowywane w WeakReference przez odwzorowanie. Czynnikiem uruchamiającym sprzątanie jest sytuacja, w której klucz nie jest już używany, jak pokazuje to przykład:

```
//: containers/CanonicalMapping.java
// Kontener WeakHashMap.
import java.util.*;

class Element {
  private String ident;
  public Element(String id) { ident = id; }
  public String toString() { return ident; }
  public int hashCode() { return ident.hashCode(); }
  public boolean equals(Object r) {
    return r instanceof Element &&
      ident.equals(((Element)r).ident);
  }
  protected void finalize() {
    System.out.println("Finalizacja " +
      getClass().getSimpleName() + " " + ident);
  }
}

class Key extends Element {
  public Key(String id) { super(id); }
}

class Value extends Element {
  public Value(String id) { super(id); }
}

public class CanonicalMapping {
  public static void main(String[] args) {
    int size = 1000;
    // Albo wybór rozmiaru z wiersza poleceń:
    if(args.length > 0)
      size = new Integer(args[0]);
    Key[] keys = new Key[size];
    WeakHashMap<Key,Value> map =
      new WeakHashMap<Key,Value>();
    for(int i = 0; i < size; i++) {
      Key k = new Key(Integer.toString(i));
```

```
      Value v = new Value(Integer.toString(i));
      if(i % 3 == 0)
        keys[i] = k; // Zachowanie jako "prawdziwych" referencji
      map.put(k, v);
    }
    System.gc();
  }
} /* (Execute to see output) *///:~
```

Klasa Key musi zawierać metody `hashCode()` oraz `equals()`, ponieważ jest używana jako klucz w strukturze z haszowaniem — opisałem to wcześniej w tym rozdziale.

W czasie uruchomienia programu widać, że odśmiecacz pamięci ominie każdy co trzeci klucz, gdyż zwykłe odwołania do tych kluczy zostały również zamieszczone w tablicy keys, dlatego te obiekty nie mogą być zniszczone.

Kontenery Java 1.0 i 1.1

Niestety, sporo programów zostało napisanych z wykorzystaniem kontenerów Java 1.0 i 1.1 i nawet nowe są czasami nadal tworzone z zastosowaniem tamtych klas. Tak więc, chociaż nie powinieneś nigdy stosować starych kontenerów, pisząc nowy kod, to wciąż musisz coś o nich wiedzieć. Poprzednie kontenery były dosyć ograniczone, nie da się więc dużo o nich powiedzieć (ponieważ stanowią już anachronizm, spróbuję powstrzymać się od komentowania kilku bardzo złych decyzji projektantów).

Vector i Enumeration

Jedyną samoczynnie rozszerzającą się strukturą w Java 1.0 i 1.1 był `Vector`, często więc go stosowano. Jego słabe strony są zbyt liczne, by je tutaj opisać (zajrzyj do pierwszego wydania książki dostępnej za darmo na stronie *www.MindView.net*). Zasadniczo możesz myśleć o nim jak o `ArrayList` z długimi, niezręcznymi nazwami metod. W odnowionej bibliotece kontenerów `Vector` został przystosowany tak, by pasował jako rodzaj `Collection` i `List`. Okazuje się to odrobinę przewrotne i niektóre osoby mogą pomyśleć, że `Vector` stał się lepszy, podczas gdy w rzeczywistości jest on dołączony tylko w celu obsługi kodu wcześniej napisanego z myślą o starszych wydaniach Javy.

W wersji 1.0 i 1.1 Javy wybrano nową nazwę dla iteratora — enumerator — zamiast stosować termin, z którym wszyscy już się zapoznali. Interfejs `Enumeration` przypomina interfejs `Iterator` tylko dwiema metodami i stosuje dłuższe nazwy: `boolean hasMoreElements()` daje wartość `true`, jeśli enumeracja zawiera więcej elementów, a `Object nextElement()` zwraca następny element enumeracji, jeśli takowy istnieje (w przeciwnym razie zgłasza wyjątek).

`Enumeration` jest tylko interfejsem, a nie implementacją i nawet nowe biblioteki czasami nadal stosują tę starą odmianę iteratora — co jest niefortunne, ale raczej nieszkodliwe. Mimo że powinieneś zawsze, kiedy tylko możesz, stosować `Iterator` we własnym kodzie, powinieneś być przygotowany na biblioteki, które chcą obsługiwać `Enumeration`.

Można uzyskać Enumeration dla dowolnego rodzaju Collection poprzez wywołanie metody Collections.enumeration(), jak pokazuje to przykład:

```
//: containers/Enumerations.java
// Java 1.0/1.1: Vector i Enumeration.
import java.util.*;
import net.mindview.util.*;

public class Enumerations {
  public static void main(String[] args) {
    Vector<String> v =
      new Vector<String>(Countries.names(10));
    Enumeration<String> e = v.elements();
    while(e.hasMoreElements())
      System.out.print(e.nextElement() + ", ");
    // Utworzenie enumeratora kolekcji:
    e = Collections.enumeration(new ArrayList<String>());
  }
} /* Output:
ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, BURUNDI, CAMEROON,
CAPE VERDE, CENTRAL AFRICAN REPUBLIC,
*///:~
```

By uzyskać Enumeration, wywołuje się metodę elements() i dalej można już wykorzystać go do iterowania w przód.

W ostatnim wierszu tworzona jest lista ArrayList oraz stosowana metoda enumeration() do pozyskania iteratora Enumeration z Iterator odpowiadającego liście. W ten sposób, jeśli mamy starszy kod, który żąda Enumeration, nadal można stosować nowe kontenery.

Hashtable

Jak można było zobaczyć podczas porównania wydajności, podstawowy kontener Hashtable jest bardzo podobny do HashMap, nawet pod względem nazw metod. Nie ma jednak powodów, by stosować Hashtable zamiast HashMap w nowym kodzie.

Stack

Pojęcie stosu zostało wprowadzone wcześniej, przy okazji LinkedList. Dziwne jest to, że w Java 1.0 i 1.1, zamiast wykorzystywać Vector jako klocek do budowy, Stack został *wywiedziony* z Vector. Posiada więc wszystkie właściwości i zachowanie wektora z dodatkowym zachowaniem stosu. Trudno się dowiedzieć, czy projektanci otwarcie zdecydowali, iż był to szczególnie użyteczny sposób realizacji, czy też był to po prostu skutek naiwności. Niezależnie od przyczyn, klasa ta nie została dokładnie przemyślana przed jej umieszczeniem w dystrybucyjnej wersji JDK, przez co jej fatalny projekt *wciąż* jest dostępny publicznie (absolutnie nie należy jej jednak używać).

Przedstawiam prosty przykład stosu, w którym zamieszczamy na nim każdy z elementów wyliczenia. Przykład pokazuje przy okazji, że w roli stosu można bardzo wygodnie wykorzystywać kontener LinkedList albo klasę Stack utworzoną w ramach rozdziału „Kolekcje obiektów":

```
// : containers/Stacks.java
// Klasa Stack.
import java.util.*;
import static net.mindview.util.Print.*;

enum Month { JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
  JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER }

public class Stacks {
  public static void main(String[] args) {
    Stack<String> stack = new Stack<String>();
    for(Month m : Month.values())
      stack.push(m.toString());
    print("stack = " + stack);
    // Traktowanie stosu jako kontenera Vector:
    stack.addElement("Ostatni wiersz");
    print("element 5 = " + stack.elementAt(5));
    print("zdejmowanie elementów:");
    while(!stack.empty())
      printnb(stack.pop() + " ");

    // Kontener LinkedList w roli stosu:
    LinkedList<String> lstack = new LinkedList<String>();
    for(Month m : Month.values())
      lstack.addFirst(m.toString());
    print("lstack = " + lstack);
    while(!lstack.isEmpty())
      printnb(lstack.removeFirst() + " ");

    // Klasa Stack z rozdziału
    // "Kolekcje obiektów":
    net.mindview.util.Stack<String> stack2 =
      new net.mindview.util.Stack<String>();
    for(Month m : Month.values())
      stack2.push(m.toString());
    print("stack2 = " + stack2);
    while(!stack2.empty())
      printnb(stack2.pop() + " ");

  }
} /* Output:
stack = [JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER,
OCTOBER, NOVEMBER]
element 5 = JUNE
zdejmowanie elementów:
Ostatni wiersz NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL MARCH
FEBRUARY JANUARY lstack = [NOVEMBER, OCTOBER, SEPTEMBER, AUGUST, JULY, JUNE, MAY,
APRIL, MARCH, FEBRUARY, JANUARY]
NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL MARCH FEBRUARY
JANUARY stack2 = [NOVEMBER, OCTOBER, SEPTEMBER, AUGUST, JULY, JUNE, MAY, APRIL,
MARCH, FEBRUARY, JANUARY]
NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL MARCH FEBRUARY
JANUARY
*///:~
```

Dla stałych zebranych w typie wyliczeniowym `Month` generowane są reprezentacje tekstowe w postaci obiektów `String`. Obiekty te są wstawiane na stos wywołaniem metody `push()`, a potem zdejmowane z niego wywołaniem `pop()`. By zaznaczyć pochodzenie, na

obiekcie typu Stack przeprowadzane są również operacje klasy Vector. Jest to możliwe dzięki dziedziczeniu — stos *jest* wektorem. Zatem wszystkie operacje możliwe do wykonania na Vector, mogą być przeprowadzane na Stack, jak choćby elementAt().

Jak wspominałem wcześniej, kiedy potrzebujemy zachowania stosu, powinno się stosować kontener LinkedList albo klasę net.mindview.util.Stack, utworzoną na bazie kontenera LinkedList.

BitSet

Bitset jest przydatny, kiedy chcemy sprawnie przechowywać wiele informacji dwustanowych (typu „włączone-wyłączone"). Jest to wydajne tylko z punktu widzenia rozmiaru; jeżeli poszukujesz wydajnego dostępu, to jest odrobinę wolniejszy niż użycie tablic.

Dodatkowo maksymalny rozmiar kontenera BitSet to long, czyli 64 bity. Oznacza to, że do przechowywania czegoś mniejszego niż np. 8 bitów BitSet będzie nieekonomiczny; lepiej napisać własną klasę lub po prostu użyć tablicy do przechowywania znaczników, jeśli wymagany jest inny rozmiar (dotyczy to jedynie tych sytuacji, w których zamierzamy tworzyć *dużo* wartości dwustanowych; decyzje takie należy podejmować na podstawie uprzednich pomiarów wydajności; podjęcie decyzji o własnej implementacji kontenera typu dwustanowego tylko z powodu domniemanego, a nieszkodliwego marnotrawstwa, skończy się niepotrzebnym komplikowaniem kodu i stratą czasu).

Normalny kontener rozszerza się, kiedy dołoży się więcej elementów — BitSet czyni to również. Zamieszczony przykład pokazuje BitSet w działaniu:

```
//: containers/Bits.java
// Kontener BitSet.
import java.util.*;
import static net.mindview.util.Print.*;

public class Bits {
  public static void printBitSet(BitSet b) {
    print("bity: " + b);
    StringBuilder bbits = new StringBuilder();
    for(int j = 0; j < b.size() ; j++)
      bbits.append(b.get(j) ? "1" : "0");
    print("wzorzec bitowy: " + bbits);
  }
  public static void main(String[] args) {
    Random rand = new Random(47);
    // Pobranie najmniej znaczącego bajta z nextInt():
    byte bt = (byte)rand.nextInt();
    BitSet bb = new BitSet();
    for(int i = 7; i >= 0; i--)
      if(((1 << i) & bt) != 0)
        bb.set(i);
      else
        bb.clear(i);
    print("wartość byte: " + bt);
    printBitSet(bb);

    short st = (short)rand.nextInt();
    BitSet bs = new BitSet();
```

```
    for(int i = 15; i >= 0; i--)
      if(((1 << i) &  st) != 0)
        bs.set(i);
      else
        bs.clear(i);
    print("wartość short: " + st);
    printBitSet(bs);

    int it = rand.nextInt();
    BitSet bi = new BitSet();
    for(int i = 31; i >= 0; i--)
      if(((1 << i) &  it) != 0)
        bi.set(i);
      else
        bi.clear(i);
    print("wartość int: " + it);
    printBitSet(bi);

    // testowanie zbiorów bitowych o ponad 64 bitach:
    BitSet b127 = new BitSet();
    b127.set(127);
    print("ustawienie bitu nr 127: " + b127);
    BitSet b255 = new BitSet(65);
    b255.set(255);
    print("ustawienie bitu nr 255: " + b255);
    BitSet b1023 = new BitSet(512);
    b1023.set(1023);
    b1023.set(1024);
    print("ustawienie bitu nr 1023: " + b1023);
  }
} /* Output:
wartość byte: -107
bity: {0, 2, 4, 7}
wzorzec bitowy: 10101001000000000000000000000000000000000000000000000000000000000000000
wartość short: 1302
bity: {1, 2, 4, 8, 10}
wzorzec bitowy: 01101000101000000000000000000000000000000000000000000000000000000000000
wartość int: -2014573909
bity: {0, 1, 3, 5, 7, 9, 11, 18, 19, 21, 22, 23, 24, 25, 26, 31}
wzorzec bitowy: 11010101010100000011011111100001000000000000000000000000000000000
ustawienie bitu nr 127: {127}
ustawienie bitu nr 255: {255}
ustawienie bitu nr 1023: {1023, 1024}
*///:~
```

Generator liczb losowych został wykorzystany do uzyskania losowych wartości typów `byte`, `short` oraz `int`; każda z nich jest następnie transformowana w odpowiadający wzór bitowy do `BitSet`. Wszystko działa poprawnie, ponieważ `BitSet` jest 64-bitowy, więc żaden z nich nie powoduje zwiększenia jego rozmiaru. Potem tworzony jest `BitSet` o większym rozmiarze. Konstruktor alokuje pamięć dla dwukrotnej liczby bitów. Nadal można ustawić bit 1024 lub wyższy.

W przypadku ustalonego zestawu nazwanych znaczników lepszym rozwiązaniem jest zwykle kontener `EnumSet` (zobacz rozdział „Typy wyliczeniowe"). `EnumSet` pozwala na manipulowanie nazwami, a nie numerami bitów, co minimalizuje ryzyko popełnienia błędu w odwołaniu do elementu zbioru bitowego. Do tego `EnumSet` zabezpiecza programistę przed nieumyślnym dodaniem nowych znaczników (w `BitSet` można dowolnie

ustawiać bity o coraz większych numerach — kontener sam się rozszerza), co mogłoby wprowadzać do programu groźne i trudne do wykrycia błędy. `BitSet` przejawia wyższość nad `EnumSet` jedynie tam, gdzie nie wiadomo z góry, jaka ilość znaczników będzie wykorzystywana, albo przypisywanie im nazw byłoby niezasadne, ewentualnie tam, gdzie potrzebne są specjalne funkcje kontenera `BitSet` (odsyłam tu do dokumentacji klas `EnumSet` i `BitSet` w JDK).

Podsumowanie

Biblioteka kontenerów to niewątpliwie najważniejsza biblioteka języka obiektowego. Większość zadań programistycznych wymaga takiego czy innego zastosowania kontenerów i są one wykorzystywane częściej niż jakiekolwiek inne komponenty biblioteczne. W niektórych językach programowania (choćby w Pythonie) najważniejsze komponenty kontenerowe — listy, zbiory i odwzorowania — są wbudowanymi elementami języka.

W rozdziale „Kolekcje obiektów" pokazywałem, jak można wykorzystać kontenery w ciekawych zadaniach przy minimalizacji nakładów programistycznych. Jednak w którymś momencie wiedza o podstawach stosowania kontenerów przestaje być wystarczająca — aby wykorzystać w pełni ich możliwości, trzeba ogarnąć operację haszowania (aby móc pisać własne implementacje metody `hashCode()` dla typów danych przeznaczonych do przechowywania w kontenerach haszowanych), a także wiedzieć o implementacjach poszczególnych kontenerów dostatecznie dużo, aby podejmować świadome decyzje co do ich wdrażania w danych rozwiązaniach. Stąd potrzeba wyodrębnienia osobnego rozdziału, który omawiałby te i kilka innych aspektów biblioteki kontenerów w języku Java.

Projekt biblioteki kontenerów nie jest prosty (złożoność to zresztą cecha charakterystyczna wielu problemów projektowych). W języku C++ biblioteka kontenerowa jest implementowana szeregiem osobnych klas. To podejście lepsze niż żadne, ale nienadające się do prostego przeniesienia do realiów Javy. Z drugiej strony widywałem biblioteki kontenerów, które składały się z jednej zaledwie klasy pełniącej równocześnie rolę kontenerów sekwencyjnych i asocjacyjnych. Biblioteka kontenerów w Javie zdaje się bardziej wyważona: oferuje programiście wszystko to, czego można oczekiwać od dojrzałej biblioteki kontenerów, ale łatwiej ją opanować niż jej odpowiedniki z C++ i im podobne. Efekt końcowy jest jednak w paru miejscach cokolwiek dziwaczny; w przeciwieństwie do decyzji podejmowanych w pierwszych bibliotekach Javy owe dziwactwa nie są jednak przypadkowe, a są efektami starannie rozważonych decyzji odnośnie złożoności.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 18.

Wejście-wyjście

Stworzenie dobrego systemu wejścia-wyjścia jest jednym z najtrudniejszych zadań stojących przed projektantem języka.

Najważniejszym wyzwaniem jest konieczność ujęcia wszystkich zagadnień. Nie tylko trzeba uwzględnić rozmaite źródła i ujścia wejścia i wyjścia (pliki, konsola, połączenia sieciowe), należy także komunikować się z nimi na wiele sposobów (tryb sekwencyjny, swobodnego dostępu, binarny, znakowy, przez wczytywanie wierszy, słów itd.).

Projektanci biblioteki Javy „zaatakowali" problem, tworząc dużą liczbę klas. W systemie wejścia-wyjścia (ang. *input/output*) Javy jest istotnie tyle klas, że z początku działa to odstraszająco (jak na ironię, w rzeczywistości został on zaprojektowany tak, aby właśnie zapobiec eksplozji klas). W wersjach Javy późniejszych niż 1.0 w bibliotece wejścia-wyjścia została również wprowadzona istotna zmiana — oryginalna, zorientowana bajtowo biblioteka została poszerzona o zorientowane znakowo, oparte na Unicode klasy wejścia-wyjścia. W JDK 1.4 zostały dodane klasy *nio* (skrót od „new I/O" — nowe wejście-wyjście; ta nazwa będzie używana jeszcze przez długie lata, a przecież pochodzi z Javy 1.4, więc jest już raczej nienowa) w celu poprawienia efektywności i poszerzenia możliwości funkcjonalnych. W rezultacie istnieje wiele klas wymagających poznania, zanim zrozumie się działanie biblioteki wejścia-wyjścia na tyle, aby używać jej właściwie. Dość ważne jest zrozumienie historii ewolucji biblioteki wejścia-wyjścia Javy, nawet jeśli Twoją pierwszą reakcją jest: „Przestań zanudzać tą historią, po prostu pokaż, jak tego używać!". Problem leży w tym, że bez historycznej perspektywy niektóre z klas wydadzą się niezrozumiałe, a rozstrzygnięcie, czy istnieje potrzeba ich użycia, trudne.

Ten rozdział pozwoli Ci poznać różnorodność klas wejścia-wyjścia w standardowej bibliotece Javy oraz nauczy Cię, jak ich używać.

Klasa File

Zanim poznamy klasy, które zajmują się właściwym zapisem i odczytem danych ze strumienia, przyjrzymy się narzędziu bibliotecznemu ułatwiającemu obsługę katalogów plików.

Nazwa klasy `File` jest zwodnicza — można by pomyśleć, że odnosi się ona do pliku, co nie jest prawdą. Lepszą nazwą dla klasy byłaby zapewne `FilePath` (ścieżka pliku). Klasa reprezentuje bowiem albo *nazwę* konkretnego pliku, albo *nazwę* zbioru plików w katalogu. W tym drugim przypadku o zbiór plików można zapytać, używając metody `list()`, która zwraca tablicę obiektów `String`. Zwrócenie tablicy zamiast elastycznego pojemnika ma tu sens, ponieważ liczba elementów jest ustalona, i, żeby uzyskać spis zawartości innego katalogu, trzeba po prostu utworzyć kolejny obiekt `File`. Zaprezentuję teraz przykład użycia tej klasy, łącznie ze związanym z nią interfejsem `FilenameFilter`.

Wypisywanie zawartości katalogu

Przypuśćmy, że zachodzi potrzeba wypisania listy plików znajdujących się w danym katalogu. Zawartość obiektu `File` może być wypisana na dwa sposoby. Jeżeli metoda `list()` zostanie wywołana bez argumentów, otrzymamy pełną listę plików zawartą w obiekcie. Jeśli jednak chcemy tę listę ograniczyć — na przykład do plików z rozszerzeniem *.java* — należy użyć „filtra katalogu", który jest klasą informującą, w jaki sposób wybrać obiekty do wypisania.

Poniżej znajduje się przykład. Warto zauważyć, że wyniki zostały bez dodatkowego wysiłku posortowane alfabetycznie za pomocą metody `java.util.Array.sort()` i komparatora `String.CASE_INSENSITIVE_ORDER`:

```
//: io/DirList.java
// Wypisuje zawartość katalogu wybieraną za pomocą wyrażeń regularnych.
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList {
  public static void main(String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
      list = path.list();
    else
      list = path.list(new DirFilter(args[0]));
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
      System.out.println(dirItem);
  }
}

class DirFilter implements FilenameFilter {
  private Pattern pattern;
  public DirFilter(String regex) {
    pattern = Pattern.compile(regex);
  }
  public boolean accept(File dir, String name) {
    return pattern.matcher(name).matches();
  }
} /* Output:
DirectoryDemo.java
```

```
DirList.java
DirList2.java
DirList3.java
*///:~
```

Klasa `DirFilter` implementuje interfejs `FilenameFilter`. Zobacz, jak prosty jest ten interfejs:

```
public interface FilenameFilter {
  boolean accept(File dir, String name);
}
```

Jak widać, działanie obiektów tego typu sprowadza się do dostarczenia metody o nazwie `accept()`. Utworzenie tej klasy wiązało się właśnie z koniecznością zapewnienia metody `accept()` — tak aby metoda `list()` mogła ją wywołać w celu stwierdzenia, które nazwy plików powinny zostać dołączone do wypisywanej listy. Zatem technika ta jest często przedstawiana jako *wywołanie zwrotne* (ang. *callback*). Precyzyjnie rzecz biorąc, jest to przykład wzorca projektowego *Strategy*, gdyż metoda `list()` implementuje proste możliwości funkcjonalne, a dostarczany do niej obiekt `FilenameFilter` jest *strategią* uzupełniającą algorytm działania tej metody. Ponieważ metoda `list()` przyjmuje jako argument obiekt `FilenameFilter`, oznacza to, że można przekazać do niej obiekt dowolnej klasy implementującej `FilenameFilter`, aby określić (nawet w trakcie wykonywania programu) sposób, w jaki ma działać `list()`. Celem tej techniki jest zapewnienie elastyczności działania kodu.

Metoda `accept()` może przyjmować jako argument obiekt `File` reprezentujący katalog, w którym znajduje się dany plik, oraz obiekt `String` zawierający nazwę tego pliku. Należy pamiętać, że metoda `list()` wywołuje `accept()` dla każdej nazwy pliku w katalogu, aby stwierdzić, które z nich powinny zostać dołączone do listy — decyduje o tym wartość typu `boolean` zwracana przez `accept()`.

Metoda `accept()` wykorzystuje obiekt wyrażenia regularnego `matcher`, aby sprawdzić, czy wyrażenie regularne `regex` pasuje do nazwy pliku. Dzięki użyciu metody `accept()` metoda `list()` zwraca tablicę.

Anonimowe klasy wewnętrzne

Ten przykład nadaje się świetnie do przepisania z użyciem anonimowej klasy wewnętrznej (opisanej w rozdziale „Klasy wewnętrzne"). W pierwszej kolejności tworzona jest metoda `filter()` zwracająca referencję do `FilenameFilter`:

```
//: io/DirList2.java
// Użycie anonimowych klas wewnętrznych.
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList2 {
  public static FilenameFilter filter(final String regex) {
    // Utworzenie anonimowej klasy wewnętrznej:
    return new FilenameFilter() {
      private Pattern pattern = Pattern.compile(regex);
      public boolean accept(File dir, String name) {
        return pattern.matcher(name).matches();
```

```
      }
    }; // Koniec anonimowej klasy wewnętrznej
  }
  public static void main(String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
      list = path.list();
    else
      list = path.list(filter(args[0]));
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
      System.out.println(dirItem);
  }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:~
```

Zauważmy, że argument dla metody `filter()` musi być typu `final`. Jest to wymagane przez anonimową klasę wewnętrzną, by mogła ona użyć obiektu spoza własnego zasięgu. Taki sposób zaprojektowania jest ulepszeniem, ponieważ klasa `FilenameFilter` jest teraz ściśle związana z `DirList2`. Można również posunąć się krok dalej i zdefiniować anonimową klasę wewnętrzną jako argument dla `list()` — w tym przypadku jest nawet mniejsza:

```
//: io/DirList3.java
// Budowanie anonimowej klasy wewnętrznej "miejscowo".
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList3 {
  public static void main(final String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
      list = path.list();
    else
      list = path.list(new FilenameFilter() {
        private Pattern pattern = Pattern.compile(args[0]);
        public boolean accept(File dir, String name) {
          return pattern.matcher(name).matches();
        }
      });
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
      System.out.println(dirItem);
  }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:~
```

Teraz argument przekazywany do `main()` jest typu `final`, ponieważ anonimowa klasa wewnętrzna korzysta wprost z `args[0]`.

Przykład ten pokazuje zastosowanie anonimowych klas wewnętrznych do tworzenia unikalnych klas „jednorazowego użycia" służących do rozwiązywania napotkanych problemów. Korzyścią jest zgromadzenie kodu, który służy do rozwiązania konkretnego problemu, odizolowanego w jednym miejscu. Z drugiej strony, uzyskane klasy są dosyć nieczytelne, należy więc używać go rozważnie.

Ćwiczenie 1. Zmodyfikuj przykład *DirList.java* (albo jeden z jego wariantów) tak, aby klasa `FilenameFilter` otwierała i wczytywała każdy plik (za pomocą klasy `net.mindview.util.TextFile`), a następnie akceptowała wszystkie te pliki, które zawierają którekolwiek ze słów przekazanych jako argumenty wiersza wywołania programu (3).

Ćwiczenie 2. Stwórz klasę o nazwie `SortedDirList` z konstruktorem, który przyjmuje jako argument obiekt `File` i tworzy posortowaną listę plików znajdujących się w określonym przez ten obiekt katalogu. Napisz dla klasy dwie przeciążone metody `list()`; pierwsza ma produkować całą listę, druga jej podzbiór wg kryterium podanego jako argument (który ma być wyrażeniem regularnym) (2).

Ćwiczenie 3. Zmień program *DirList.java* (albo jeden z jego wariantów) tak, aby obliczał sumę rozmiarów wybieranych plików (3).

Narzędzie do przeglądania katalogów

Powszechnym zadaniem programistycznym jest przetwarzanie zbiorów plików, czy to wybieranych z katalogu lokalnego, czy też z całego poddrzewa katalogów. Przydałoby się więc narzędzie, które wygeneruje taki zbiór. Poniższa klasa narzędziowa tworzy tablicę obiektów `File` reprezentujących obiekty katalogu lokalnego za pomocą metody `local()` albo — za pomocą metody `walk()` — listę `List<File>` całego drzewa katalogów zakorzenionego w podanym katalogu (obiekty `File` są użyteczniejsze od nazw plików, bo zawierają dodatkowe informacje o plikach). Pliki są wybierane do zbioru na podstawie wyrażenia regularnego podawanego przez programistę-klienta:

```
// : net/mindview/util/Directory.java
// Generuje sekwencję obiektów File pasujących do wyrażenia
// regularnego, a wybieranych z katalogu bieżącego albo
// poddrzewa katalogów.
package net.mindview.util;
import java.util.regex.*;
import java.io.*;
import java.util.*;

public final class Directory {
  public static File[]
  local(File dir, final String regex) {
    return dir.listFiles(new FilenameFilter() {
      private Pattern pattern = Pattern.compile(regex);
      public boolean accept(File dir, String name) {
        return pattern.matcher(
          new File(name).getName()).matches();
      }
    });
```

```
  }
  public static File[]
  local(String path, final String regex) { // Przeciążona
    return local(new File(path), regex);
  }
  // Klasa do zwracania par obiektów:
  public static class TreeInfo implements Iterable<File> {
    public List<File> files = new ArrayList<File>();
    public List<File> dirs = new ArrayList<File>();
    // Domyślny element iteracji to lista File:
    public Iterator<File> iterator() {
      return files.iterator();
    }
    void addAll(TreeInfo other) {
      files.addAll(other.files);
      dirs.addAll(other.dirs);
    }
    public String toString() {
      return "katalogi: " + PPrint.pformat(dirs) +
        "\n\npliki: " + PPrint.pformat(files);
    }
  }
  public static TreeInfo
  walk(String start, String regex) { // Początek rekurencji
    return recurseDirs(new File(start), regex);
  }
  public static TreeInfo
  walk(File start, String regex) { // Przeciążona
    return recurseDirs(start, regex);
  }
  public static TreeInfo walk(File start) { // Dawaj wszystko
    return recurseDirs(start, ".*");
  }
  public static TreeInfo walk(String start) {
    return recurseDirs(new File(start), ".*");
  }
  static TreeInfo recurseDirs(File startDir, String regex){
    TreeInfo result = new TreeInfo();
    for(File item : startDir.listFiles()) {
      if(item.isDirectory()) {
        result.dirs.add(item);
        result.addAll(recurseDirs(item, regex));
      } else // Zwykły plik
        if(item.getName().matches(regex))
          result.files.add(item);
    }
    return result;
  }
  // Prosta walidacja:
  public static void main(String[] args) {
    if(args.length == 0)
      System.out.println(walk("."));
    else
      for(String arg : args)
       System.out.println(walk(arg));
  }
} ///:~
```

Metoda `local()` wykorzystuje wariant wywołania `File.list()` o nazwie `listFiles()`, zwracający tablicę obiektów `File`. Widać też użycie filtra `FilenameFilter`. Gdyby zamiast tablicy potrzebny był kontener `List`, można by skonwertować wynik wywołaniem metody `Arrays.asList()`.

Metoda `walk()` konwertuje nazwę katalogu początkowego na obiekt typu `File` i wywołuje metodę `recurseDirs()`, która podejmuje rekurencyjny spacer po podkatalogach połączony ze zbieraniem informacji o ich zawartości. Dla odróżnienia plików zwykłych od katalogów wartością zwracaną jest krotka (para) obiektów — kontener `List` przechowujący zwykłe pliki i drugi, przechowujący katalogi. Pola pary są celowo oznaczone jako publiczne, bo zadaniem `TreeInfo` jest jedynie zgrupowanie obiektów — gdybyśmy w ramach wartości zwracanej zwracali wprost `List`, nie uczynilibyśmy jej przecież prywatną. Klasa `TreeInfo` implementuje interfejs `Iterable<File>`, który domyślnie przegląda obiekty plików; chcąc przeglądać listę katalogów, musimy się posłużyć wywołaniem metody `iterator()` z kwalifikacją `dirs`..

Metoda `TreeInfo.toString()` używa klasy `PPrint`, która dba o czytelność wyjścia. Domyślna implementacja `toString()` dla kontenerów wypisuje wszystkie elementy kontenera w jednym wierszu; w przypadku obszerniejszych zbiorów plików taki wypis byłby nieczytelny, stąd możliwość formatowania alternatywnego. Służy do tego właśnie `PPrint`, jako narzędzie uzupełniające o znaki nowego wiersza i wcięcia przy każdym elemencie:

```
//: net/mindview/util/PPrint.java
// Klasa formatująca wypisywanie zawartości kolekcji
package net.mindview.util;
import java.util.*;

public class PPrint {
  public static String pformat(Collection<?> c) {
    if(c.size() == 0) return "[]";
    StringBuilder result = new StringBuilder("[");
    for(Object elem : c) {
      if(c.size() != 1)
        result.append("\n  ");
      result.append(elem);
    }
    if(c.size() != 1)
      result.append("\n");
    result.append("]");
    return result.toString();
  }
  public static void pprint(Collection<?> c) {
    System.out.println(pformat(c));
  }
  public static void pprint(Object[] c) {
    System.out.println(pformat(Arrays.asList(c)));
  }
} ///:~
```

Metoda `pformat()` generuje sformatowany ciąg znaków reprezentujący zawartość kolekcji, a metoda `pprint()` wykorzystuje `pformat()` do wypisania tego ciągu na wyjściu. Zauważ, że przypadki braku elementów w kontenerze i obecności pojedynczego elementu są obsługiwane specjalnie. Jest też wersja `pprint()` dla tablic.

Narzędzie w postaci klasy `Directory` znajduje się w pakiecie `net.mindview.util`, jest więc łatwo dostępne. Oto przykład jego użycia:

```
//: io/DirectoryDemo.java
// Próbka możliwości klasy Directory.
import java.io.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DirectoryDemo {
  public static void main(String[] args) {
    // Wszystkie katalogi:
    PPrint.pprint(Directory.walk(".").dirs);
    // Wszystkie pliki o nazwach zaczynających się od 'T'
    for(File file : Directory.local(".", "T.*"))
      print(file);
    print("----------------------");
    // Wszystkie pliki źródłowe języka Java zaczynające się na 'T':
    for(File file : Directory.walk(".", "T.*\\.java"))
      print(file);
    print("======================");
    // Pliki klas zawierające w nazwie 'Z' bądź 'z':
    for(File file : Directory.walk(".",".*[Zz].*\\.class"))
      print(file);
  }
} /* Output: (Sample)
[.\xfiles]
.\TestEOF.class
.\TestEOF.java
.\TransferTo.class
.\TransferTo.java
----------------------
.\TestEOF.java
.\TransferTo.java
.\xfiles\ThawAlien.java
======================
.\FreezeAlien.class
.\GZIPcompress.class
.\ZipCompress.class
*///:~
```

W razie potrzeby stosowanie wyrażeń regularnych możesz sobie przypomnieć, wracając do rozdziału „Ciągi znaków" — zapewne ułatwi Ci to ogarnięcie znaczenia drugiego argumentu wywołań metod `local()` i `walk()`.

Możemy pójść o krok dalej i utworzyć narzędzie, które będzie przeglądać katalogi i w miarę przeglądania przetwarzać dopasowane pliki — znów implementując wzorzec projektowy *Strategy*:

```
//: net/mindview/util/ProcessFiles.java
package net.mindview.util;
import java.io.*;

public class ProcessFiles {
  public interface Strategy {
    void process(File file);
  }
  private Strategy strategy;
```

```
  private String ext;
  public ProcessFiles(Strategy strategy, String ext) {
    this.strategy = strategy;
    this.ext = ext;
  }
  public void start(String[] args) {
    try {
      if(args.length == 0)
        processDirectoryTree(new File("."));
      else
        for(String arg : args) {
          File fileArg = new File(arg);
          if(fileArg.isDirectory())
            processDirectoryTree(fileArg);
          else {
            // Pozwala użytkownikowi na pomijanie rozszerzenia:
            if(!arg.endsWith("." + ext))
              arg += "." + ext;
            strategy.process(
              new File(arg).getCanonicalFile());
          }
        }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  public void
  processDirectoryTree(File root) throws IOException {
    for(File file : Directory.walk(
        root.getAbsolutePath(), ".*\\." + ext))
      strategy.process(file.getCanonicalFile());
  }
  // A tak się tego używa:
  public static void main(String[] args) {
    new ProcessFiles(new ProcessFiles.Strategy() {
      public void process(File file) {
        System.out.println(file);
      }
    }, "java").start(args);
  }
} /* (Execute to see output) *///:~
```

Interfejs `Strategy` jest zagnieżdżony w klasie `ProcessFiles`, więc aby go zaimplementować, trzeba skorzystać z zapisu `implements ProcessFiles.Strategy`, co przy okazji stanowi dla czytającego kod dodatkowy opis kontekstu implementacji. Klasa `ProcessFiles` zajmuje się wyszukiwaniem plików o zadanym rozszerzeniu (argument `ext` konstruktora), a w razie dopasowania pliku przekazuje go po prostu do obiektu `Strategy` (który również jest argumentem konstruktora klasy `ProcessFiles`).

Jeśli konstruktor nie otrzyma żadnych argumentów, założy, że chcemy przejrzeć wszystkie podkatalogi katalogu bieżącego. Można też ograniczyć poszukiwanie do konkretnego pliku, z rozszerzeniem nazwy lub bez niego, a także przeszukać jeden albo kilka katalogów.

W metodzie `main()` mamy prosty przykład użycia nowego narzędzia; wypisuje on nazwy wszystkich plików kodu źródłowego w języku Java zgodnie z argumentami wywołania.

Ćwiczenie 4. Użyj metody `Directory.walk()` do podsumowania rozmiarów plików w drzewie katalogów; do sumowania mają być wybierane tylko pliki o nazwach pasujących do podanego wyrażenia regularnego (2).

Ćwiczenie 5. Zmodyfikuj program *ProcessFiles.java* tak, aby dopasowywał nazwy plików na podstawie wyrażenia regularnego, a nie prostego rozszerzenia nazwy (1).

Tworzenie katalogów i sprawdzanie ich obecności

Klasa `File` jest czymś więcej niż tylko reprezentacją istniejącego pliku lub katalogu. Za pomocą obiektu `File` można również utworzyć katalog lub całą ścieżkę katalogową, jeśli taka nie istnieje. Mamy także możliwość wglądu we właściwości plików (rozmiar, data ostatniej modyfikacji, prawa dostępu), sprawdzenia, czy obiekt `File` reprezentuje plik czy katalog, oraz usunięcia pliku. Następujący program pokazuje wykorzystanie niektórych innych metod dostępnych w klasie `File` (pełny zestaw można znaleźć w dokumentacji JDK publikowanej w witrynie *http://java.sun.com*):

```
//: io/MakeDirectories.java
// Demonstruje użycie klasy File do tworzenia
// katalogów i manipulowania plikami.
// {Args: MakeDirectoriesTest}
import java.io.*;

public class MakeDirectories {
  private static void usage() {
    System.err.println(
      "Stosowanie:MakeDirectories ścieżka1 ...\n" +
      "Tworzy wymienione ścieżki\n" +
      "Stosowanie:MakeDirectories -d ścieżka1 ...\n" +
      "Usuwa wymienione ścieżki\n" +
      "Stosowanie:MakeDirectories -r ścieżka1 ścieżka2\n" +
      "Zmiana nazwy ze ścieżka1 na ścieżka2");
    System.exit(1);
  }
  private static void fileData(File f) {
    System.out.println(
      "Ścieżka bezwzględna: " + f.getAbsolutePath() +
      "\n Do odczytu: " + f.canRead() +
      "\n Do zapisu: " + f.canWrite() +
      "\n Nazwa: " + f.getName() +
      "\n Nadrzędny: " + f.getParent() +
      "\n Ścieżka: " + f.getPath() +
      "\n rozmiar: " + f.length() +
      "\n Ostatnia modyfikacja: " + f.lastModified());
    if(f.isFile())
      System.out.println("To plik");
    else if(f.isDirectory())
      System.out.println("To katalog");
  }
  public static void main(String[] args) {
    if(args.length < 1) usage();
    if(args[0].equals("-r")) {
      if(args.length != 3) usage();
      File
        old = new File(args[1]),
        rname = new File(args[2]);
```

```
      old.renameTo(rname);
      fileData(old);
      fileData(rname);
      return; // Wyjście z metody main
    }
    int count = 0;
    boolean del = false;
    if(args[0].equals("-d")) {
      count++;
      del = true;
    }
    count--;
    while(++count < args.length) {
      File f = new File(args[count]);
      if(f.exists()) {
        System.out.println(f + " istnieje");
        if(del) {
          System.out.println("usuwanie..." + f);
          f.delete();
        }
      }
      else { // Nie istnieje
        if(!del) {
          f.mkdirs();
          System.out.println("utworzono " + f);
        }
      }
      fileData(f);
    }
  }
} /* Output: (80% match)
utworzono MakeDirectoriesTest
Ścieżka bezwzględna: E:\TIJ4\code\io\MakeDirectoriesTest
 Do odczytu: true
 Do zapisu: true
 Nazwa: MakeDirectoriesTest
 Nadrzędny: null
 Ścieżka: MakeDirectoriesTest
 rozmiar: 0
 Ostatnia modyfikacja: 1142504694000
To katalog
*///:~
```

W funkcji `fileData()` widać zastosowanie różnych metod uzyskiwania informacji o pliku lub ścieżce dostępu.

Pierwszą metodą wykorzystywaną przez `main()` jest `renameTo()`, która pozwala na zmianę nazwy (lub przeniesienie) pliku według nowej ścieżki reprezentowanej przez argument, który również jest obiektem `File`. Działa to również dla ścieżek dostępu do katalogów dowolnej długości.

Eksperymentując z powyższym programem, przekonamy się, że można stworzyć ścieżki katalogowe dowolnej złożoności — cała praca wykonywana jest przez metodę `mkdirs()`.

Ćwiczenie 6. Użyj klasy `ProcessFiles` do wyszukania wszystkich plików kodu źródłowego Java w wybranym poddrzewie katalogu; program ma wybrać pliki, które zostały zmodyfikowane po określonej dacie (5).

Wejście i wyjście

Programistyczne biblioteki wejścia-wyjścia często wykorzystują abstrakcyjne pojęcie *strumienia* (ang. *stream*), reprezentującego dowolne źródło lub ujście danych, jako obiektu zdolnego do wysyłania lub odbierania porcji danych. Strumień ukrywa szczegóły tego, co dzieje się z danymi wewnątrz konkretnego urządzenia wejścia-wyjścia.

Klasy biblioteki wejścia-wyjścia w Javie są podzielone według wejścia (ang. *input*) i wyjścia (ang. *output*), o czym można się przekonać, przeglądając hierarchię klas Javy w dokumentacji JDK. Przez dziedziczenie wszystkie klasy wyprowadzone z klasy `InputStream` lub `Reader` mają metody podstawowe `read()` służące do odczytania jednego bajta lub tablicy bajtów. Podobnie wszystkie klasy dziedziczące po klasie `OutputStream` lub `Writer` mają podstawowe metody `write()` do zapisu pojedynczego bajta lub tablicy bajtów. W rzeczywistości nie zachodzi potrzeba użycia tych metod bezpośrednio — istnieją, aby mogły z nich korzystać inne klasy, dostarczające bardziej użyteczny interfejs. Dlatego rzadko tworzy się obiekt strumieniowy, używając pojedynczej klasy, ale raczej nawarstwia się kilka obiektów, aby zapewnić pożądaną funkcję. Konieczność stworzenia więcej niż jednego obiektu, aby uzyskać w efekcie pojedynczy obiekt strumieniowy, jest podstawowym powodem tego, że biblioteka wejścia-wyjścia Javy może na początku sprawić wiele kłopotów.

Pomocne jest podzielenie klas na kategorie ze względu na pełnione przez nie funkcje. Projektanci biblioteki wejścia-wyjścia w Java 1.0 podjęli decyzję, że klasy mające cokolwiek wspólnego z obsługą wejścia będą dziedziczyły po klasie `InputStream`, natomiast z klasy `OutputStream` wywodzą się funkcje związane z obsługą wyjścia.

Zgodnie z praktyką stosowaną w tej książce przedstawię teraz przegląd klas, po szczegóły — takie jak wyczerpująca lista metod — odsyłając Cię jednak do pełnej dokumentacji dostępnej w internecie.

Typy InputStream

Zadaniem `InputStream` jest reprezentacja klas dostarczających na wejście dane z różnych źródeł. Mogą nimi być:

1. Tablica bajtów;
2. Obiekt `String`;
3. Plik;
4. „Potok" (ang. *pipe* — działa jak rzeczywisty potok — rzeczy włożone do źródła pojawiają się przy ujściu);
5. Sekwencja innych strumieni, które można zebrać do jednego;
6. Inne źródła, takie jak połączenie internetowe (Informacje na ten temat można znaleźć w książce *Thinking in Enterprise Java*, dostępnej w witrynie *www.MindView.net*).

Z każdym z tych źródeł jest związana odpowiednia podklasa InputStream. W dodatku typem InputStream jest również klasą FilterInputStream dostarczająca klasę bazową dla klas „dekoratorów", które dołączają atrybuty lub użyteczne interfejsy do strumieni wejściowych. Będzie o tym mowa później (tabela 18.1).

Tabela 18.1. *Typy InputStream*

Klasa	Działanie	Argumenty konstruktora / Sposób użycia
ByteArrayInputStream	Jako InputStream używany jest bufor (tablica) w pamięci	Tablica, z której mają być odczytywane bajty
		Jako źródło danych. Należy połączyć z obiektem FilterInputStream, aby zapewnić użyteczny interfejs
StringBufferInputStream	Konwertuje String do InputStream	Obiekt typu String. Wewnętrzna implementacja wykorzystuje obiekt StringBuffer
		Jako źródło danych. Należy połączyć z obiektem FilterInputStream, aby zapewnić użyteczny interfejs
FileInputStream	Odczytuje informacje z pliku	String reprezentujący nazwę pliku albo obiekt typu File lub FileDescriptor
		Jako źródło danych. Należy połączyć z obiektem FilterInputStream, aby zapewnić użyteczny interfejs
PipedInputStream	Odzyskuje dane zapisywane przez połączony z nim potok — obiekt PipedOutputStream	Obiekt klasy PipedOutputStream
		Jako źródło danych w aplikacjach wielowątkowych. Należy połączyć z obiektem FilterInputStream, aby zapewnić użyteczny interfejs
SequenceInputStream	Konwertuje dwa lub więcej obiektów InputStream do pojedynczego InputStream	Dwa obiekty InputStream lub obiekt Enumeration dla kontenera obiektów InputStream
		Jako źródło danych. Należy połączyć z obiektem FilterInputStream, aby zapewnić użyteczny interfejs
FilterInputStream	Abstrakcyjna klasa służąca jako interfejs dla klas „dekoratorów" dostarczających dodatkowe możliwości innym klasom InputStream. Patrz tabela 12.3	Patrz tabela 18.3
		Patrz tabela 18.3

Typy OutputStream

Ta kategoria zawiera klasy decydujące o tym, na jakie wyjście kierowane są dane: tablica bajtów (ale nie String; można go stworzyć, posługując się tablicą bajtów), plik lub „potok".

Klasa `FilterOutputStream` udostępnia klasę bazową dla klas „dekoratorów" dołączających do strumieni wyjściowych dodatkowe atrybuty lub bardziej użyteczny interfejs. Mowa jest o tym w dalszej części rozdziału (tabela 18.2).

Tabela 18.2. *Typy OutputStream*

Klasa	Działanie	Argumenty konstruktora Sposób użycia
`ByteArrayOutputStream`	Tworzy bufor w pamięci. Wszystkie dane wysyłane do strumienia umieszczane są w tym buforze	Opcjonalnie rozmiar początkowy bufora Wskazuje przeznaczenie danych. Należy połączyć z obiektem `FilterOutputStream`, aby zapewnić użyteczny interfejs
`FileOutputStream`	Wysyła informacje do pliku	`String` reprezentujący nazwę pliku albo obiekt typu `File` lub `FileDescriptor` Wskazuje przeznaczenie danych. Należy połączyć z obiektem `FilterOutputStream`, aby zapewnić użyteczny interfejs
`PipedOutputStream`	Wpisywane informacje automatycznie trafiają na wejście do połączonego z nim potoku — obiektu `PipedInputStream`	Obiekt typu `PipedInputStream` Wskazuje przeznaczenie danych w aplikacjach wielowątkowych. Należy połączyć z obiektem `FilterOutputStream`, aby zapewnić użyteczny interfejs
`FilterOutputStream`	Abstrakcyjna klasa służąca jako interfejs dla klas „dekoratorów" dostarczających dodatkowe możliwości innym klasom `OutputStream`. Patrz tabela 18.4	Patrz tabela 18.4 Patrz tabela 18.4

Dodawanie atrybutów i użytecznych interfejsów

Dekoratory zostały przedstawione w rozdziale „Typy ogólne". Biblioteka wejścia-wyjścia Javy wymaga wielu rozmaitych kombinacji poszczególnych funkcji, co jest powodem, dla którego zostały zastosowane dekoratory[1]. To właśnie jest przyczyną istnienia klas „filtrów" w bibliotece wejścia-wyjścia Javy: abstrakcyjna klasa „filtra" jest klasą bazową dla wszystkich dekoratorów (dekorator musi mieć taki sam interfejs, jak obiekt, który dekoruje, ale może też ten interfejs rozszerzać i tak dzieje się w kilku przypadkach klas „filtrów").

Stosowanie tego wzorca ma jednak pewną wadę. Dekoratory pozwalają na dużo większą elastyczność w programowaniu — ceną jest jednak zwiększenie złożoności kodu. Powodem tego, że biblioteka wejścia-wyjścia Javy jest kłopotliwa w użyciu, jest konieczność stworzenia wielu klas — „rdzennych" typów wejścia-wyjścia oraz wszystkich dekoratorów — aby otrzymać pojedynczy obiekt wejścia-wyjścia o pożądanych właściwościach.

[1] Nie wiadomo czy ta decyzja projektowa była dobra, zwłaszcza w porównaniu z prostotą bibliotek wejścia-wyjścia w innych językach programowania. Jednak tak właśnie się ją usprawiedliwia.

Klasy zapewniające interfejs dekoratora dla konkretnych obiektów `InputStream` i `OutputStream` to odpowiednio `FilterInputStream` i `FilterOutputStream` — nie są to zbyt intuicyjne nazwy. `FilterInputStream` i `FilterOutputStream` są klasami wywodzącymi się od podstawowych klas biblioteki wejścia-wyjścia, `InputStream` i `OutputStream`, co zapewnia spełnienie kluczowego dla dekoratorów wymagania, by miały wspólny interfejs z dekorowanymi przez siebie obiektami.

Odczyt z InputStream za pomocą FilterInputStream

Klasy `FilterInputStream` pozwalają na osiągnięcie dwóch odmiennych celów. `DataInputStream` umożliwia odczytywanie danych różnych typów podstawowych, jak również obiektów klasy `String` (wszystkie metody zaczynają się od „read", np. `readByte()`, `readFloat()` itd.). To, w połączeniu z klasą `DataOutputStream`, pozwala przenosić dane typów podstawowych z jednego miejsca w inne za pośrednictwem strumieni. „Miejsca" te są określone przez klasy w tabeli 18.1.

Pozostałe klasy `FilterInputStream` modyfikują wewnętrzny sposób działania `InputStream`: czy jest buforowany, czy śledzi numerację odczytywanych wierszy (pozwalając zapytać o numer wiersza lub go ustawić) i czy można wstawić z powrotem do strumienia pojedynczy znak. Ostatnie dwie klasy wykorzystywane są głównie przy budowie kompilatora (zostały zapewne dodane jako element eksperymentu polegającego na konstrukcji kompilatora języka Java w tymże języku) i raczej nie zachodzi potrzeba ich użycia w zwykłym programowaniu.

Najczęściej dobrym rozwiązaniem jest buforowanie wejścia, niezależnie od rodzaju urządzenia wejścia-wyjścia, z którym nawiązujemy połączenie, zatem wydaje się, że w bibliotece wejścia-wyjścia byłoby sensowniejszym rozwiązaniem traktować jako specjalny przypadek raczej użycie niebuforowanego niż buforowanego wejścia — odwrotnie niż ma to miejsce w bibliotece wejścia-wyjścia Javy (tabela 18.3).

Tabela 18.3. *Typy FilterInputStream*

Klasa	Działanie	Argumenty konstruktora Sposób użycia
`DataInputStream`	Używany łącznie z `DataOutputStream` umożliwia odczytywanie danych typów podstawowych ze strumienia w sposób niezależny od platformy systemowej	`InputStream` Zawiera pełny interfejs pozwalający na odczytywanie danych typów podstawowych
`BufferedInputStream`	Używany, aby zapobiec fizycznemu odczytowi za każdym żądaniem kolejnych danych	`InputStream`, opcjonalnie rozmiar bufora Nie dostarcza *własnego* interfejsu, jedynie wymusza użycie bufora. Należy dołączyć interfejs
`LineNumberInputStream`	Śledzi numery wierszy w strumieniu wejścia; umożliwia wywołanie `getLineNumber()` i `setLineNumber(int)`	`InputStream` Zapewnia jedynie numerowanie wierszy, więc prawdopodobnie zajdzie potrzeba dołączenia interfejsu

Tabela 18.3. *Typy FilterInputStream — ciąg dalszy*

Klasa	Działanie	Argumenty konstruktora Sposób użycia
PushbackInputStream	Zapewnia jednobajtowy bufor zwrotny, aby można było zwrócić do strumienia ostatni odczytany znak	InputStream Używana przez kompilator na etapie analizy leksykalnej kodu. Raczej nie stosowana w programowaniu

Zapis do OutputStream za pomocą FilterOutputStream

Klasą uzupełniającą do DataInputStream jest DataOutputStream, formatująca typy podstawowe i obiekty String do strumienia w taki sposób, że każdy obiekt DataInputStream uruchomiony na dowolnej platformie sprzętowej może je odczytać. Wszystkie metody zaczynają się od „write", np. writeByte(), writeFloat() itd.

Oryginalnie klasa PrintStream została stworzona, aby umożliwić wypisywanie wszystkich podstawowych typów danych w formacie nadającym się do przeglądania, w przeciwieństwie do DataOutputStream, której celem jest umieszczenie danych w strumieniu w taki sposób, aby mogły zostać zrekonstruowane przez DataInputStream.

Dwoma ważnymi metodami w PrintStream są print() oraz println(). Zostały one przeciążone tak, aby móc wypisywać rozmaite typy danych. Różnica między nimi polega na tym, że println() dodaje do wypisywanego ciągu znaków znak nowego wiersza.

Stosowanie klasy PrintStream przysparza nieco problemów, ponieważ jej metody przechwytują wszystkie wyjątki typu IOException, które mogą się zdarzyć (musisz więc jawnie sprawdzić, czy wystąpił błąd, wywołując metodę checkError() — jeśli błąd wystąpił, zwróci ona wartość true). Oprócz tego PrintStream nie radzi sobie dobrze z wyświetlaniem zlokalizowanych komunikatów, a także nie obsługuje znaków końca wierszy niezależnie od platformy (te błędy zostały na szczęście usunięte w bliźniaczej klasie PrintWriter).

BufferedOutputStream jest modyfikatorem nakazującym strumieniowi korzystanie z bufora tak, że fizyczny zapis nie następuje za każdym razem, gdy dane są wysyłane do strumienia. Można sądzić, że klasa ta będzie wykorzystywana zawsze w przypadku generacji jakichkolwiek danych wyjściowych (tabela 18.4).

Tabela 18.4. *Typy FilterOutputStream*

Klasa	Działanie	Argumenty konstruktora Sposób użycia
DataOutputStream	Używany łącznie z DataInputStream umożliwia zapisywanie podstawowych typów danych do strumienia w sposób niezależny od platformy systemowej	OutputStream Zawiera pełny interfejs pozwalający na zapisywanie podstawowych typów danych

Tabela 18.4. *Typy FilterOutputStream — ciąg dalszy*

Klasa	Działanie	Argumenty konstruktora Sposób użycia
`PrintStream`	Formatuje dane wyjściowe. Podczas gdy `DataOutputStream` steruje przechowywaniem danych, `PrintStream` decyduje o sposobie ich wypisywania	`OutputStream`, opcjonalnie wartość `boolean` wskazująca, czy bufor ma być opróżniany z każdym znakiem nowego wiersza Powinien być końcową klasą opakowującą obiekt `OutputStream`. Prawdopodobnie użyjesz go wiele razy
`BufferedOutputStream`	Zapobiega fizycznemu zapisowi za każdym kolejnym żądaniem danych. Aby „przepchnąć" zawartość bufora, można wywołać `flush()`	`OutputStream`, opcjonalnie rozmiar bufora Nie dostarcza własnego interfejsu, jedynie uzupełnia proces o buforowanie. Należy dołączyć interfejs

Klasy Reader i Writer

Java 1.1 przyniosła ze sobą znaczącą modyfikację w bibliotece strumieni wejścia-wyjścia (w Java 2 nie nastąpiły już żadne fundamentalne zmiany). Patrząc na nowe klasy `Reader` i `Writer`, można by z początku pomyśleć, że zostały one wprowadzone, aby zastąpić klasy `InputStream` i `OutputStream`. Chodzi jednak o coś innego. Chociaż stosowanie niektórych funkcji oryginalnej biblioteki nie jest już zalecane (w razie ich użycia kompilator wystosuje odpowiednie ostrzeżenie), klasy `InputStream` i `OutputStream` wciąż dostarczają cenny zestaw funkcji dla operacji wejścia-wyjścia zorientowanych *bajtowo*, podczas gdy klasy `Reader` i `Writer` zapewniają zgodną z Unicode, zorientowaną *znakowo* obsługę wejścia-wyjścia. Oprócz tego:

7. W Java 1.1 do hierarchii `InputStream` i `OutputStream` dodane zostały nowe klasy, oczywiste jest więc, że stare klasy nie zostały zastąpione.
8. Zdarzają się przypadki wymagające użycia klas z hierarchii „bajtowej" *w kombinacji* z klasami z hierarchii „znakowej". Do osiągnięcia tego celu wprowadzono klasy „pomostowe": `InputStreamReader` konwertuje obiekt `InputStream` do `Reader`, natomiast `OutputStreamWriter` konwertuje `OutputStream` do `Writer`.

Najważniejszym powodem stworzenia hierarchii klas `Reader` i `Writer` jest dążenie do umożliwienia obsługi standardu międzynarodowego. Stara hierarchia klas strumieniowych wejścia-wyjścia obsługuje jedynie strumienie 8-bitowe i nie daje sobie rady z 16-bitowymi znakami Unicode. Ponieważ Unicode jest standardem międzynarodowym (a typ Javy `char` to właśnie 16-bitowy znak Unicode), zostały dodane nowe klasy, aby Unicode był poprawnie obsługiwany we wszystkich operacjach wejścia-wyjścia. Poza tym nowe biblioteki zostały zaprojektowane tak, aby operacje wykonywane za ich pomocą były szybsze.

Źródła i ujścia danych

Niemal wszystkie oryginalne klasy Javy, operujące na strumieniach wejścia-wyjścia, mają swoje odpowiedniki w hierarchii `Reader` lub `Writer`, zapewniające możliwość manipulacji znakami Unicode. Są jednak przypadki, kiedy właściwym rozwiązaniem jest właśnie użycie *bajtowo* zorientowanych obiektów `InputStream` i `OutputStream`; szczególnie biblioteki pakietu `java.util.zip` są zorientowane raczej bajtowo niż znakowo. Najsensowniejszym wyjściem wydaje się *próba* używania klas `Reader` i `Writer`. Przypadki wymagające zastosowania bibliotek zorientowanych bajtowo zostaną szybko wykryte — po prostu kod nie zostanie skompilowany.

Poniższa tabela pokazuje związki pomiędzy źródłami a ujściami informacji (czyli skąd fizycznie pochodzą dane i gdzie trafiają) w obu hierarchiach klas.

Źródła i ujścia

Klasy Java 1.0	Odpowiednie klasy Java 1.1
`InputStream`	`Reader`
	klasa adaptera: `InputStreamReader`
`OutputStream`	`Writer`
	klasa adaptera: `OutputStreamWriter`
`FileInputStream`	`FileReader`
`FileOutputStream`	`FileWriter`
`StringBufferInputStream` (zarzucona)	`StringReader`
(brak odpowiednika)	`StringWriter`
`ByteArrayInputStream`	`CharArrayReader`
`ByteArrayOutputStream`	`CharArrayWriter`
`PipedInputStream`	`PipedReader`
`PipedOutputStream`	`PipedWriter`

Można stwierdzić, że interfejsy tych dwóch hierarchii są podobne, jeśli nie identyczne.

Modyfikacja zachowania strumienia

W hierarchii `InputStream` i `OutputStream` strumienie były adaptowane do konkretnych potrzeb przez użycie „dekoratorów" — podklasy `FilterInputStream` i `FilterOutputStream`. Hierarchie klas `Reader` i `Writer` również korzystają z tej ideologii, ale nie do końca.

W kolejnej tabeli podobieństwo klas jest mniej widoczne niż w poprzedniej. Różnica ta spowodowana jest organizacją klas: podczas gdy `BufferedOutputStream` jest podklasą `FilterOutputStream`, `BufferedWriter` *nie jest* podklasą `FilterWriter` (która, choć abstrakcyjna, nie posiada w ogóle podklas i wygląda na to, że została zamieszczona jedynie w celu wypełnienia miejsca — albo żeby nikt się nie zastanawiał, dlaczego jej nie ma). Tym niemniej interfejsy klas są całkiem podobne.

Filtry	
Klasy Java 1.0	**Odpowiadające klasy Java 1.1**
`FilterInputStream`	`FilterReader`
`FilterOutputStream`	`FilterWriter` (klasa abstrakcyjna, nie posiada podklas)
`BufferedInputStream`	`BufferedReader` (również ma metodę `readLine()`)
`BufferedOutputStream`	`BufferedWriter`
`DataInputStream`	Należy używać `DataInputStream` (chyba że potrzebna jest `readLine()` — wtedy trzeba użyć `BufferedReader`)
`PrintStream`	`PrintWriter`
`LineNumberInputStream` (odrzucona)	`LineNumberReader`
`StreamTokenizer`	`StreamTokenizer` (należy użyć konstruktora przyjmującego obiekt `Reader`)
`PushBackInputStream`	`PushBackReader`

Jedno zalecenie jest jasne: w razie potrzeby skorzystania z metody `readLine()` nie należy robić tego więcej za pośrednictwem `DataInputStream` (ostrzega zresztą o tym kompilator), ale użyć klasy `BufferedReader`. W innych przypadkach `DataInputStream` jest nadal preferowanym do użycia składnikiem biblioteki wejścia-wyjścia.

Aby przejście do używania klasy `PrintWriter` było łatwiejsze, została ona wyposażona w konstruktory przyjmujące zarówno dowolny obiekt `OutputStream`, jak i `Writer`. Interfejs formatowania klasy `PrintWriter` jest praktycznie identyczny z interfejsem `PrintStream`.

W Javie SE5 pojawiły się dodatkowe konstruktory klasy `PrintWriter` upraszczające tworzenie plików przy wypisywaniu danych na wyjście — wkrótce z nich skorzystamy.

Konstruktor `PrintWriter` ma również opcję automatycznego opróżniania bufora — dzieje się to przy każdym użyciu metody `println()`, jeśli znacznik konstruktora został ustawiony.

Klasy niezmienione

Niektóre klasy nie uległy zmianie w Java 1.1 w stosunku do Java 1.0:

Klasy Java 1.0, które pozostały nie zmienione w Java 1.1
`DataOutputStream`
`File`
`RandomAccessFile`
`SequenceInputStream`

Szczególnie bez zmian jest używana klasa `DataOutputStream`, tak więc do przechowywania danych w formacie niezależnym od platformy systemowej należy używać hierarchii `InputStream` i `OutputStream`.

Osobna i samodzielna RandomAccessFile

Klasa `RandomAccessFile` jest używana do obsługi plików zawierających rekordy o znanym rozmiarze, tak aby można się było między nimi przemieszczać za pomocą metody `seek()`, czytać je lub zmieniać. Nie muszą one mieć tego samego rozmiaru: wystarczy jeśli będzie się dało określić, jak są duże i gdzie dokładnie znajdują się w pliku.

Z początku trudno jest uwierzyć, że `RandomAccessFile` nie jest częścią hierarchii `InputStream` lub `OutputStream`. W rzeczywistości nie ma ona z nimi nic wspólnego — poza tym, że implementuje interfejsy `DataInput` i `DataOutput` (implementują je także klasy `DataInputStream` i `DataOutputStream`). Nie używa także zestawu funkcji istniejących klas `InputStream` i `OutputStream` — jest zupełnie oddzielną klasą, napisaną od zera ze wszystkimi swoimi metodami. Powodem takiej sytuacji jest zachowanie klasy `RandomAccessFile`, zasadniczo odmienne od działania pozostałych typów wejścia-wyjścia, ponieważ w pliku możemy się poruszać zarówno w przód, jak i do tyłu. W każdym razie jest to całkowicie samodzielna klasa, wywodząca się wprost z klasy `Object`.

Zasadniczo `RandomAccessFile` działa jak `DataInputStream` połączona z `DataOutputStream` oraz z dodatkowymi metodami — `getFilePointer()` (aby sprawdzić, gdzie wewnątrz pliku się znajdujemy), `seek()` (aby przemieścić się do innego miejsca) i `length()` (aby stwierdzić, jaka jest maksymalna długość pliku). Oprócz tego konstruktor wymaga drugiego argumentu (tak samo jak `fopen()` w C) wskazującego, czy chcemy tylko swobodnie odczytywać plik (`"r"`) czy czytać i pisać do niego (`"rw"`). Nie ma obsługi plików tylko do zapisu (ang. *write-only*), co mogłoby sugerować, że `RandomAccessFile` mogłaby pracować równie dobrze, gdyby dziedziczyła po `DataInputStream`.

Jedynie w `RandomAccessFile` dostępne są metody służące do przeszukiwania i pracują one tylko z plikami. Co prawda `BufferedInputStream` pozwala na zaznaczenie (`mark()`) pozycji (o której informacja jest przechowywana w pojedynczej zmiennej wewnętrznej), a potem powrót (`reset()`) do tej pozycji, jednak funkcja ta jest ograniczona i niezbyt przydatna.

Wiele, albo niemal wszystkie możliwości klasy `RandomAccessFile` zostały zastąpione w *nio* przez *pliki odwzorowywane w pamięci* (ang. *memory-mapped files*), które zostaną opisane w dalszej części rozdziału.

Typowe zastosowania strumieni wejścia-wyjścia

Chociaż klasy strumieni wejścia-wyjścia można łączyć ze sobą na wiele sposobów, najczęściej używa się tylko kilku kombinacji. Poniższe przykłady mogą być wykorzystywane jako punkt odniesienia dla typowych operacji wejścia-wyjścia.

W przykładach tych obsługa wyjątków zostanie uproszczona do postaci zakładającej przekazywanie wyjątków na konsolę, ale to metoda odpowiednia jedynie w najprostszych programach i narzędziach. W poważniejszych programach należałoby zadbać o konkretną obsługę błędów.

Buforowany plik wejścia

Aby otworzyć plik dla wejścia znakowego, należy użyć klasy `FileReader` z obiektem `String` albo `File` jako nazwą pliku. W celu szybszego działania dobrze jest buforować plik, zatem przekazujemy otrzymaną referencję do konstruktora klasy `BufferedReader`. Ponieważ `BufferedReader` również dostarcza metodę `readLine()`, jest to już ostateczny obiekt i interfejs, z którego będą odczytywane dane. W przypadku osiągnięcia końca pliku `readLine()` zwraca `null`.

```
//: io/BufferedInputFile.java
import java.io.*;

public class BufferedInputFile {
  // Zrzuca wyjątki na konsolę:
  public static String
  read(String filename) throws IOException {
    // Wypisanie wejścia wierszami:
    BufferedReader in = new BufferedReader(
      new FileReader(filename));
    String s;
    StringBuilder sb = new StringBuilder();
    while((s = in.readLine())!= null)
      sb.append(s + "\n");
    in.close();
    return sb.toString();
  }
  public static void main(String[] args)
  throws IOException {
    System.out.print(read("BufferedInputFile.java"));
  }
} /* (Execute to see output) *///:~
```

Obiekt `sb` klasy `StringBuilder` kumuluje zawartość pliku (wraz ze znakami nowych wierszy, które trzeba dodawać ręcznie, bo metoda `readLine()` je obcina). Na koniec plik jest zamykany wywołaniem metody `close()`[2].

Ćwiczenie 7. Otwórz plik tekstowy tak, aby móc odczytywać wiersz po wierszu. Odczytuj każdy wiersz jako ciąg `String` i umieszczaj go na liście `LinkedList`. Wypisz wszystkie elementy `LinkedList` w odwrotnym porządku (2).

Ćwiczenie 8. Zmodyfikuj ćwiczenie 7. tak, aby nazwa odczytywanego pliku podawana była jako argument z wiersza poleceń (1).

Ćwiczenie 9. Zmodyfikuj ćwiczenie 8. tak, aby przetworzyć wszystkie ciągi znaków z `LinkedList` na wielkie litery i wysłać wyniki do *System.out* (1).

Ćwiczenie 10. Zmodyfikuj ćwiczenie 8. tak, aby program przyjmował jako argumenty z wiersza poleceń słowa do znalezienia w pliku. Wypisz wszystkie wiersze, w których wystąpią żądane słowa (2).

[2] Pierwotnie metoda `close()` była przewidziana do automatycznego wywoływania w ramach `finalize()`; zobaczysz zresztą niedługo metody `finalize()` klas wejścia-wyjścia zdefiniowane z użyciem metody `close()`. Jednak z dyskusji prezentowanej w innych rozdziałach wynika, że `finalize()` nie zawsze działa zgodnie z zamierzeniami architektów języka, więc najbezpieczniej jest wywoływać `close()` jawnie.

Ćwiczenie 11. W przykładzie *innerclasses/GreenhouseController.java* klasa `GreenhouseController` zawiera sztywno zapisany w kodzie zestaw zdarzeń. Zmień program tak, aby wczytywał zdarzenia i ich nazwy z pliku tekstowego (2). Zastosuj wzorzec projektowy *Factory Method* do konstrukcji zdarzeń — zobacz *Thinking in Patterns (in Java)* na stronach *www.MindView.net* (8).

Wejście z pamięci

Tym razem obiekt `String` zwracany z `BufferedInputFile.read()` zostanie wykorzystany do utworzenia obiektu `StringReader`. Potem za pomocą metody `read()` odczytujemy kolejno pojedyncze znaki i wysyłamy na konsolę.

```
//: io/MemoryInput.java
import java.io.*;

public class MemoryInput {
  public static void main(String[] args)
  throws IOException {
    StringReader in = new StringReader(
      BufferedInputFile.read("MemoryInput.java"));
    int c;
    while((c = in.read()) != -1)
      System.out.print((char)c);
  }
} /* (Execute to see output) *///:~
```

Zauważmy, że `read()` zwraca kolejny bajt jako `int` i należy go zrzutować na `char`, aby został poprawnie wypisany.

Formatowane wejście z pamięci

Aby czytać dane „sformatowane", należy użyć `DataInputStream` klasy wejścia-wyjścia zorientowanej bajtowo (a nie znakowo). Dlatego używamy klas z hierarchii `InputStream`, a nie `Reader`. Oczywiście wszystkie dane (takie jak plik) można odczytać jako ciąg bajtów, używając klas `InputStream`, jednak tu mamy do czynienia z obiektem `String`.

```
//: io/FormattedMemoryInput.java
import java.io.*;

public class FormattedMemoryInput {
  public static void main(String[] args)
  throws IOException {
    try {
      DataInputStream in = new DataInputStream(
        new ByteArrayInputStream(
          BufferedInputFile.read(
            "FormattedMemoryInput.java").getBytes()));
      while(true)
        System.out.print((char)in.readByte());
    } catch(EOFException e) {
      System.err.println("Koniec strumienia");
    }
  }
} /* (Execute to see output) *///:~
```

`ByteArrayInputStream` wymaga przekazania tablicy bajtów. Aby ją wygenerować, wykorzystujemy metodę `getBytes()` klasy `String`. Powstały obiekt klasy `ByteArrayInputStream` stanowi właściwy obiekt `InputStream`, nadający się do przekazania do `DataInputStream`.

Jeżeli odczytujemy znaki z `DataInputStream` po jednym bajcie za pomocą metody `readByte()`, każda wartość może być prawidłowym rezultatem, zatem zwracany wynik nie może przesądzić o tym, czy napotkaliśmy już koniec pliku. Zamiast tego można użyć metody `available()`, aby sprawdzić, ile jeszcze znaków jest dostępnych. Poniższy przykład pokazuje, jak czytać po jednym bajcie z pliku:

```
//: io/TestEOF.java
// Wykrywanie końca pliku przy odczycie bajt po bajcie.
import java.io.*;

public class TestEOF {
  public static void main(String[] args)
  throws IOException {
    DataInputStream in = new DataInputStream(
      new BufferedInputStream(
        new FileInputStream("TestEOF.java")));
    while(in.available() != 0)
      System.out.print((char)in.readByte());
  }
} /* (Execute to see output) *///:~
```

Metoda `available()` działa w różny sposób w zależności od źródła, z którego odczytujemy dane; dosłownie jest to „liczba bajtów, które mogą być odczytane *bez blokowania*". W przypadku pliku będzie to cały plik, jednak w przypadku innych strumieni nie musi to być prawdą, zatem lepiej używać jej w sposób przemyślany.

W takich przypadkach można również wykryć koniec wejścia, przechwytując wyjątek. Jednak generalnie wykorzystywanie wyjątków w celu kontroli przepływu uważane jest za nadużywanie tego mechanizmu.

Wyjście do pliku

Operację zapisu danych do pliku realizuje klasa `FileWriter`. Praktycznie zawsze warto opakować taki obiekt w `BufferedWriter` (można usunąć to opakowanie i zbadać różnicę w wydajności — buforowanie bardzo zwiększa szybkość operacji wejścia-wyjścia). Potem całość jest dekorowana obiektem `PrintWriter` zajmującym się formatowaniem. Plik utworzony w ten sposób nadaje się do czytania, tak jak zwykły plik tekstowy:

```
//: io/BasicFileOutput.java
import java.io.*;

public class BasicFileOutput {
  static String file = "BasicFileOutput.out";
  public static void main(String[] args)
  throws IOException {
    BufferedReader in = new BufferedReader(
      new StringReader(
        BufferedInputFile.read("BasicFileOutput.java")));
    PrintWriter out = new PrintWriter(
      new BufferedWriter(new FileWriter(file)));
```

```
    int lineCount = 1;
    String s;
    while((s = in.readLine()) != null )
      out.println(lineCount++ + ": " + s);
    out.close();
    // Wypisanie zawartości zapisanego pliku:
    System.out.println(BufferedInputFile.read(file));
  }
} /* (Execute to see output) *///:~
```

Wiersze zapisywane do pliku są uzupełniane kolejnymi numerami. Zauważ, że nie używamy klasy `LineNumberReader`, bo nie ma takiej potrzeby. Za pomocą prostego licznika w postaci zmiennej można w łatwy sposób śledzić numerację wierszy.

Kiedy strumień wejściowy zostaje wyczerpany, `readLine()` zwróci wartość `null`. Metoda `close()` dla `out` wywoływana jest w sposób jawny, ponieważ w przeciwnym razie mogłoby się okazać, że bufor nie został opróżniony i plik jest niekompletny.

Uproszczony zapis do pliku

Java SE5 uzupełniła klasę `PrintWriter` dodatkowym, pomocniczym konstruktorem, dzięki któremu nie trzeba ręcznie dokonywać dekoracji za każdym razem, kiedy chce się utworzyć plik tekstowy i zapisać do niego dane. Oto poprzedni przykład (*BasicFileOutput.java*) przepisany z użyciem tego udogodnienia:

```
//: io/FileOutputShortcut.java
import java.io.*;

public class FileOutputShortcut {
  static String file = "FileOutputShortcut.out";
  public static void main(String[] args)
  throws IOException {
    BufferedReader in = new BufferedReader(
      new StringReader(
       BufferedInputFile.read("FileOutputShortcut.java")));
    // A oto uproszczenie:
    PrintWriter out = new PrintWriter(file);
    int lineCount = 1;
    String s;
    while((s = in.readLine()) != null )
      out.println(lineCount++ + ": " + s);
    out.close();
    // Wypisanie zawartości zapisanego pliku:
    System.out.println(BufferedInputFile.read(file));
  }
} /* (Execute to see output) *///:~
```

Nie pozbawiamy się w ten sposób buforowania, zyskujemy za to wygodę. Niestety, nie wszystkie powszechnie wykonywane operacje doczekały się takiego skrócenia, więc typowa operacja wejścia-wyjścia wciąż wymaga nadmiernej ilości kodu. Jednak dzięki prezentowanej nieco dalej klasie `TextFile` (wykorzystywanej już wcześniej) owe typowe zadania można znacznie uprościć.

Ćwiczenie 12. Zmodyfikuj ćwiczenie 8., by program otwierał również plik tekstowy i aby można w nim było zapisywać dane tekstowe. Wypisz do niego wiersze z kontenera `LinkedList`, uzupełniając je numerami (nie próbuj używać klas `LineNumber`...) (3).

Ćwiczenie 13. Zmodyfikuj program *BasicFileOutput.java* tak, aby do śledzenia licznika wierszy wykorzystywał klasę `LineNumberReader`. Zauważ, że znacznie łatwiej jest samodzielnie śledzić numery wierszy (3).

Ćwiczenie 14. Na podstawie pliku *BasicFileOutput.java* napisz program, który porówna wydajność zapisu do pliku przy zastosowaniu buforowania i bez niego (2).

Przechowywanie i odzyskiwanie danych

`PrintWriter` formatuje dane tak, aby nadawały się do czytania przez człowieka. Natomiast, aby wysłać na wyjście dane w sposób czytelny dla innego strumienia, należy użyć strumienia `DataOutputStream` do zapisu, a `DataInputStream` do odzyskania danych. Oczywiście źródło tych strumieni mogłoby być dowolne, tu jednak używamy pliku z buforowaniem odczytu i zapisu. `DataOutputStream` i `DataInputStream` są zorientowane bajtowo i dlatego wymagają współpracy z klasami hierarchii `OutputStream` i `InputStream`:

```
//: io/StoringAndRecoveringData.java
import java.io.*;

public class StoringAndRecoveringData {
  public static void main(String[] args)
  throws IOException {
    DataOutputStream out = new DataOutputStream(
      new BufferedOutputStream(
        new FileOutputStream("Data.txt")));
    out.writeDouble(3.14159);
    out.writeUTF("To była liczba pi");
    out.writeDouble(1.41413);
    out.writeUTF("A to pierwiastek kwadratowy liczby 2");
    out.close();
    DataInputStream in = new DataInputStream(
      new BufferedInputStream(
        new FileInputStream("Data.txt")));
    System.out.println(in.readDouble());
    // Ciąg kodowany wedle Java-UTF odtworzy
    // poprawnie jedynie metoda readUTF():
    System.out.println(in.readUTF());
    System.out.println(in.readDouble());
    System.out.println(in.readUTF());
  }
} /* Output:
3.14159
To była liczba pi
1.41413
A to pierwiastek kwadratowy liczby 2
*///:~
```

W przypadku użycia `DataOutputStream` do zapisu danych Java gwarantuje, że dane będą prawidłowo odczytane przy zastosowaniu `DataInputStream` — niezależnie od tego, na jakich platformach chcemy dokonywać odczytu i zapisu. Ma to nieocenioną wartość —

zrozumie ją każdy, kto tracił czas, zastanawiając się nad kłopotami związanymi ze specyficznym dla danego systemu sposobem prezentacji danych. Ten problem znika, jeśli dysponujemy Javą na obu platformach[3].

Jeśli używamy `DataOutputStream`, jedynym wiarygodnym sposobem zapisu obiektu `String` tak, aby mógł zostać odczytany przez `DataInputStream`, jest zastosowanie kodowania UTF-8, co osiągamy, stosując metody `writeUTF()` oraz `readUTF()`. UTF-8 to kodowanie wielobajtowe, w którym używany rozmiar kodowania zależy od przyjętego zestawu znaków. Jeśli pracuje się tylko (lub przeważnie) ze znakami ASCII, które zajmują jedynie 7 bitów, kodowanie Unicode to duże marnotrawstwo miejsca i (lub) pasma przesyłowego. UTF-8 natomiast koduje znaki ASCII w pojedynczym bajcie, pozostałe zaś w dwóch lub trzech. Oprócz tego długość ciągu jest przechowywana w dwóch pierwszych bajtach. Należy zwrócić uwagę na to, że metody `readUTF()` i `writeUTF()` używają specjalnego wariantu UTF-8 dla Javy (który został dokładnie opisany w dokumentacji JDK dla tych metod) i próby odczytania ciągu zapisanego za pomocą `writeUTF()` przez program nienapisany w Javie wymagają stworzenia dodatkowego kodu, aby ciąg został odtworzony poprawnie.

Jeśli korzysta się z `writeUTF()` i `readUTF()`, można na przemian zapisywać obiekty `String` i inne typy danych, mając świadomość, że ciągi będą przechowane prawidłowo jako Unicode i łatwo będzie można je odzyskać, używając `DataInputStream`.

Metoda `writeDouble()` zapisuje do strumienia liczbę typu `double`, natomiast komplementarna metoda `readDouble()` pozwala na jej odczytanie (istnieją analogiczne metody służące do odczytu i zapisu pozostałych typów danych). Aby jednak metody odczytujące działały poprawnie, musi być znane dokładne położenie fragmentu danych w strumieniu, ponieważ liczba `double` mogłaby zostać równie dobrze odczytana jako sekwencja bajtów albo `char` itd. Dlatego albo należy przyjąć stały format danych w pliku, albo przechowywać dodatkową informację, z której można najpierw odczytać, gdzie dokładnie umieszczone są dane. W takim układzie należy rozważyć, czy do zapisywania i odtwarzania skomplikowanych struktur danych nie zastosować serializacji obiektów albo zapisu w języku XML (obie techniki są omawiane w dalszej części rozdziału).

Ćwiczenie 15. Odszukaj w JDK dokumentację klas `DataOutputStream` i `DataInputStream`. Na bazie programu *StoringAndRecoveringData.java* napisz program, który zapisze, a potem odtworzy wartości wszystkich możliwych typów obsługiwanych przez klasy `DataOutputStream` i `DaataInputStream`. Sprawdź, czy odczyt zapisanych wartości był dokładny (4).

Odczyt i zapis do plików o dostępie swobodnym

Użycie klasy `RandomAccessFile` jest podobne do użycia połączonych `DataInputStream` i `DataOutputStream`, ponieważ implementuje ona odpowiednie interfejsy: `DataInput` i `DataOutput`. Dysponujemy również metodą `seek()`, aby przemieszczać się w pliku i zmieniać poszczególne wartości.

[3] Innym sposobem rozwiązania problemu przenoszenia danych między różnymi systemami jest zastosowanie XML-a — niezależnie już od obecności Javy po obu stronach komunikacji. XML zostanie omówiony w dalszej części rozdziału.

Korzystanie z klasy RandomAccessFile wymaga znajomości układu danych w pliku. Klasa RandomAccessFile udostępnia specyficzne metody do odczytu i zapisu wartości podstawowych i ciągów UTF-8. Oto przykład:

```
//: io/UsingRandomAccessFile.java
import java.io.*;

public class UsingRandomAccessFile {
  static String file = "rtest.dat";
  static void display() throws IOException {
    RandomAccessFile rf = new RandomAccessFile(file, "r");
    for(int i = 0; i < 7; i++)
      System.out.println(
        "Wartość " + i + ": " + rf.readDouble());
    System.out.println(rf.readUTF());
    rf.close();
  }
  public static void main(String[] args)
  throws IOException {
    RandomAccessFile rf = new RandomAccessFile(file, "rw");
    for(int i = 0; i < 7; i++)
      rf.writeDouble(i*1.414);
    rf.writeUTF("Koniec pliku");
    rf.close();
    display();
    rf = new RandomAccessFile(file, "rw");
    rf.seek(5*8);
    rf.writeDouble(47.0001);
    rf.close();
    display();
  }
} /* Output:
Wartość 0: 0.0
Wartość 1: 1.414
Wartość 2: 2.828
Wartość 3: 4.242
Wartość 4: 5.656
Wartość 5: 7.069999999999999
Wartość 6: 8.484
Koniec pliku
Wartość 0: 0.0
Wartość 1: 1.414
Wartość 2: 2.828
Wartość 3: 4.242
Wartość 4: 5.656
Wartość 5: 47.0001
Wartość 6: 8.484
Koniec pliku
*///:~
```

Metoda display() otwiera plik i wypisuje z niego siedem wartości double. W metodzie main() następuje utworzenie i wypełnienie pliku, a następnie jego ponowne otwarcie i zmodyfikowanie zawartości. Ponieważ reprezentacja wartości double ma zawsze osiem bajtów, w metodzie seek(), chcąc odnaleźć piątą wartość double, podajemy pozycję bajtową 5*8.

Jak już powiedziałem, klasa `RandomAccessFile` jest niemal zupełnie wyizolowana z hierarchii klas biblioteki wejścia-wyjścia, poza faktem, że implementuje interfejsy `DataInput` i `DataOutput`. Nie obsługuje dekoracji, więc nie da się połączyć jej możliwości z możliwościami podklasy `InputStream` i `OutputStream`. Trzeba po prostu przyjąć, że `RandomAccessFile` jest należycie buforowana, ponieważ nie można jej samodzielnie narzucić tej właściwości.

Jedyna możliwość wyboru, jaką mamy w tym przypadku, to drugi argument konstruktora: `RandomAccessFile` można otworzyć tylko do odczytu (`"r"`) bądź do odczytu i zapisu (`"rw"`).

Można tu się zastanowić nad zastąpieniem obiektów `RandomAccessFile` plikami odwzorowywanymi w pamięci.

Ćwiczenie 16. Poszukaj w dokumentacji JDK wpisu dla klasy `RandomAccessFile`. Na bazie programu *UsingRandomAccessFile.java* napisz program, który zapisze do pliku, a potem wczyta z niego wszelkie możliwe wartości obsługiwane przez klasę `RandomAccessFile`. Sprawdź, czy odczyt zapisanych wartości przebiega bez zakłóceń (2).

Strumienie-potoki

Klasy `PipedInputStream`, `PipedOutputStream`, `PipedReader` i `PipedWriter` zostały jedynie wspomniane w tym rozdziale. Nie oznacza to, że nie są one użyteczne, ale ich wartość nie jest oczywista, zanim nie zrozumie się idei wielowątkowości, ponieważ ten typ strumieni wykorzystywany jest w komunikacji między wątkami. Dokładne omówienie wraz z przykładem znajduje się w rozdziale „Współbieżność".

Narzędzia do zapisu i odczytu danych z plików

Bardzo często spotykanym zadaniem programistycznym jest wczytanie zawartości pliku do pamięci, modyfikacja jej i powtórny zapis do pliku. Jednym z problemów przysparzanych przez standardową bibliotekę wejścia-wyjścia Javy jest konieczność napisania całkiem sporego kodu, aby wykonać to zadanie — nie ma żadnych funkcji pomocniczych, które mogłyby to zrobić za nas. Co gorsze, konieczność stosowania dekoratorów utrudnia zapamiętanie sposobu otwierania plików. Dlatego też dodanie do własnej biblioteki klas, które ułatwiałyby wykonywanie tych podstawowych czynności, jest bardzo sensowne. W Javie SE5 klasa `PrintWriter` otrzymała co prawda dodatkowy konstruktor, pozwalający na wygodne otwieranie plików tekstowych do zapisu, ale wciąż zostaje cały szereg pozostałych typowych zadań wejścia-wyjścia, przy których również chcielibyśmy wyeliminować konieczność pisania nadmiarowego kodu.

Poniżej przedstawiłem klasę `TextFile` wykorzystywaną już wcześniej do upraszczania odczytu i zapisu plików. Klasa zawiera statyczną metodę umożliwiającą odczyt i zapis zawartości pliku tekstowego w postaci jednego ciągu znaków. Możesz utworzyć obiekt `TextFile` przechowujący poszczególne wiersze pliku tekstowego w kontenerze `ArrayList` (dzięki czemu, operując na zawartości pliku, można korzystać z bogactwa funkcji klasy `ArrayList`):

```
// : net/mindview/util/TextFile.java
// Statyczne metody odczytu i zapisu plików tekstowych pojedynczymi
//   ciągami, z reprezentowaniem zawartości pliku jako kontenera ArrayList.
package net.mindview.util;
import java.io.*;
import java.util.*;

public class TextFile extends ArrayList<String> {
  // Odczyt pliku jak pojedynczego ciągu:
  public static String read(String fileName) {
    StringBuilder sb = new StringBuilder();
    try {
      BufferedReader in= new BufferedReader(new FileReader(
        new File(fileName).getAbsoluteFile()));
      try {
        String s;
        while((s = in.readLine()) != null) {
          sb.append(s);
          sb.append("\n");
        }
      } finally {
        in.close();
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    return sb.toString();
  }
  // Zapis pojedynczego pliku jednym wywołaniem metody:
  public static void write(String fileName, String text) {
    try {
      PrintWriter out = new PrintWriter(
        new File(fileName).getAbsoluteFile());
      try {
        out.print(text);
      } finally {
        out.close();
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  // Wczytanie pliku i podział zawartości wedle
  // dowolnego wyrażenia regularnego:
  public TextFile(String fileName, String splitter) {
    super(Arrays.asList(read(fileName).split(splitter)));
    // Wersja split() z wyrażeniem regularnym często
    // pozostawia na pierwszej pozycji ciąg pusty:
    if(get(0).equals("")) remove(0);
  }
  // Klasyczny odczyt pliku wierszami:
  public TextFile(String fileName) {
    this(fileName, "\n");
  }
  public void write(String fileName) {
    try {
      PrintWriter out = new PrintWriter(
        new File(fileName).getAbsoluteFile());
```

```
      try {
        for(String item : this)
          out.println(item);
      } finally {
        out.close();
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  // Prosty test:
  public static void main(String[] args) {
    String file = read("TextFile.java");
    write("test.txt", file);
    TextFile text = new TextFile("test.txt");
    text.write("test2.txt");
    // Podział na posortowaną listę słów (bez duplikatów):
    TreeSet<String> words = new TreeSet<String>(
      new TextFile("TextFile.java", "\\W+"));
    // Wypisanie słów zaczynających się od wielkich liter:
    System.out.println(words.headSet("a"));
  }
} /* Output:
[0, ArrayList, Arrays, Break, BufferedReader, BufferedWriter, Clean, Display, File, FileReader, FileWriter,
IOException, Klasyczny, Normally, Odczyt, Output, Podzia, PrintWriter, Prosty, Read, Regular,
RuntimeException, Simple, Static, Statyczne, String, StringBuilder, System, TextFile, Tools, TreeSet, W,
Wczytanie, Wersja, Write, Wypisanie, Zapis]
*///:~
```

Metoda `read()` dodaje poszczególne wiersze, uzupełnione o znaki nowego wiersza, do obiektu `StringBuffer`. W ten sposób tworzony jest ciąg `String` reprezentujący całą zawartość pliku. Metoda `write()` otwiera plik i zapisuje w nim cały ciąg znaków.

Zauważ, że każdorazowo otwarciu pliku towarzyszy ochrona przed wyjątkami, a także wywołanie metody `close()` w ramach klauzuli `finally` — dzięki temu mamy gwarancję zamknięcia każdego otwieranego pliku.

Konstruktor klasy odczytuje całą zawartość pliku w formie jednego ciągu znaków przy użyciu metody `read()`, a następnie, posługując się metodą `String.split()`, dzieli ją na poszczególne wiersze w miejscach wystąpienia znaków nowego wiersza. (Jeśli planujesz częste korzystanie z tej klasy, możesz zmodyfikować jej konstruktor, aby poprawić efektywność jego działania.) Klasa nie udostępnia żadnej metody „łączącej", a zatem niestatyczna metoda `write()` musi kolejno zapisywać w pliku poszczególne wiersze jego zawartości.

Ponieważ prezentowana klasa ma maksymalnie upraszczać proces odczytu i zapisu plików, wszelkie wyjątki są konwertowane na wyjątki `RuntimeException`, więc użytkownik nie musi ujmować wywołań metod klasy w blokach kodu chronionego. W razie potrzeby można jednak utworzyć wersję propagującą wyjątki do wywołującego.

Metoda `main()` przeprowadza proste sprawdzenie, aby upewnić się, że wszystkie metody działają poprawnie.

Choć kod przedstawionej klasy nie jest zbyt rozbudowany, może ona oszczędzić nam wiele czasu i ułatwić życie, o czym sam się przekonasz w dalszej części rozdziału.

Innym sposobem rozwiązania problemu wczytywania plików tekstowych jest zastosowanie klasy `java.util.Scanner`, nowości wprowadzonej w Javie SE5. Nadaje się jednak ona wyłącznie do odczytu plików, a nie ich zapisu, zresztą narzędzie to (umieszczone *poza* biblioteką `java.io`) służy głównie do tworzenia skanerów własnych języków programowania.

Ćwiczenie 17. Wykorzystując klasy `TextFile` i `Map<Character,Integer>`, napisz program, który zliczy w pliku wystąpienia poszczególnych liter alfabetu (tak, aby dla pliku zawierającego dwanaście wystąpień litery 'a' kontener `Map` zawierał element o kluczu 'a' i wartości '12') (4).

Ćwiczenie 18. Zmień plik *TextFile.java* tak, aby klasa przekazywała przechwytywane wyjątki do wywołującego (1).

Odczyt plików binarnych

Poniższe narzędzie, podobne do *TextFile.java*, ma upraszczać operacje odczytu plików binarnych:

```
// : net/mindview/util/BinaryFile.java
// Narzędzie do wczytywania danych binarnych z plików.
package net.mindview.util;
import java.io.*;

public class BinaryFile {
  public static byte[] read(File bFile) throws IOException{
    BufferedInputStream bf = new BufferedInputStream(
      new FileInputStream(bFile));
    try {
      byte[] data = new byte[bf.available()];
      bf.read(data);
      return data;
    } finally {
      bf.close();
    }
  }
  public static byte[]
  read(String bFile) throws IOException {
    return read(new File(bFile).getAbsoluteFile());
  }
} ///:~
```

Jedyna (za to przeciążona) metoda klasy przyjmuje argument typu `File`; druga wersja przyjmuje obiekt `String` określający nazwę pliku. Obie zwracają tablicę bajtów.

Metoda `available()` służy do określenia odpowiedniego rozmiaru tablicy; wykorzystywana wersja przeciążonej metody `read()` wypełnia utworzoną tablicę.

Ćwiczenie 19. Używając klas `BinaryFile` i `Map<Byte,Integer>`, napisz program, który zliczy w pliku wystąpienia bajtów o poszczególnych wartościach (2).

Ćwiczenie 20. Za pomocą metody `Directory.walk()` i klasy `BinaryFile` sprawdź, czy wszystkie pliki *.class* z drzewa katalogów zaczynają się od szesnastkowych wartości 'CAFEBABE' (4).

Standardowe wejście-wyjście

Termin *standardowe wejście-wyjście* (ang. *standard I/O*) odnosi się do koncepcji wywodzącej się z Uniksa, polegającej na wykorzystywaniu przez program pojedynczego strumienia informacji (koncepcja ta została powielona w różnych formach w wielu systemach operacyjnych, również w Windows). Wszystkie dane wejściowe programu mogą pochodzić ze *standardowego wejścia* (ang. *standard input*), dane wyjściowe wysyłane są na *standardowe wyjście* (ang. *standard output*), natomiast komunikaty o błędach trafiają na *standardowe wyjście diagnostyczne* (ang. *standard error*). Zaletą standardowego wejścia-wyjścia jest to, że programy mogą być łatwo łączone kaskadowo, tak aby standardowe wyjście z jednego było standardowym wejściem dla następnego programu. Takie kaskady to potężne narzędzia.

Czytanie ze standardowego wejścia

Dostosowując się do modelu standardowego wejścia-wyjścia, Java dostarcza trzy obiekty: `System.in`, `System.out` oraz `System.err`. W tej książce wielokrotnie już wypisywaliśmy komunikaty na standardowe wyjście za pomocą `System.out`, który oryginalnie jest już opakowany w obiekt `PrintStream`, podobnie jak `System.err`. `System.in` jest jednak czystym obiektem `InputStream`, bez opakowania. Oznacza to, że podczas gdy `System.out` i `System.err` można używać bezpośrednio, `System.in` musi zostać opakowany, zanim można będzie z niego odczytywać dane.

Przeważnie chcemy odczytywać wejście wierszami, używając `readLine()`, zatem należałoby opakować `System.in` w obiekt `BufferedReader`. Aby to zrobić, trzeba skonwertować `System.in` do obiektu `Reader`, używając `InputStreamReader`. W poniższym przykładzie wiersze wprowadzane z klawiatury są po prostu wypisywane na konsolę:

```
//: io/Echo.java
// Jak wczytywać dane ze standardowego wejścia.
// {RunByHand}
import java.io.*;

public class Echo {
  public static void main(String[] args)
  throws IOException {
    BufferedReader stdin = new BufferedReader(
      new InputStreamReader(System.in));
    String s;
    while((s = stdin.readLine()) != null && s.length()!= 0)
      System.out.println(s);
    // Pusty wiersz albo kombinacja Ctrl+Z przerywa program
  }
} ///:~
```

W deklaracji metody `main()` został wyspecyfikowany wyjątek, ponieważ metoda `readLine()` może zgłosić wyjątek `IOException`. Strumień `System.in` powinien być buforowany, tak jak większość innych strumieni.

Ćwiczenie 21. Napisz program, który przyjmie dane ze standardowego wejścia i zamieni wszystkie litery na ich wielkie odpowiedniki, po czym wypisze tak przetworzone dane na standardowym wyjściu. Przekieruj do programu zawartość wybranego pliku (technika przekierowania zależy od systemu operacyjnego) (1).

Zamiana System.out na PrintWriter

`System.out` jest obiektem klasy `PrintStream`, a w szczególności klasy `OutputStream`. Konstruktor `PrintWriter` przyjmuje jako argument obiekt `OutputStream`. Dlatego można skonwertować `System.out` do obiektu `PrintWriter` za pomocą następującego konstruktora:

```
//: io/ChangeSystemOut.java
// Zamienia System.out na PrintWriter.
import java.io.*;

public class ChangeSystemOut {
  public static void main(String[] args) {
    PrintWriter out = new PrintWriter(System.out, true);
    out.println("Ahoj, tam");
  }
} /* Output:
Hello, world
*///:~
```

Ważne jest, aby użyć dwuargumentowej wersji konstruktora klasy `PrintWriter` i jako drugi argument podać wartość `true`, która powoduje automatyczne opróżnianie bufora — w przeciwnym razie na wyjściu może się nic nie pojawić.

Przekierowywanie standardowego wejścia-wyjścia

Klasa Javy `System` pozwala na przekierowanie strumieni standardowego wejścia, wyjścia i wyjścia błędu za pośrednictwem prostych wywołań statycznych metod:

- `setIn(InputStream)`
- `setOut(PrintStream)`
- `setErr(PrintStream)`

Przekierowanie wyjścia może być szczególnie wygodne, jeśli na ekran wysyłanych jest tak dużo informacji, że nie nadąża się z ich odczytywaniem[4]. Przekierowanie wejścia natomiast jest cenne dla programów korzystających z wiersza poleceń, kiedy chcemy przetestować wielokrotne podanie tej samej sekwencji przez użytkownika. Poniższy prosty przykład pokazuje zastosowanie obu tych metod:

```
//: io/Redirecting.java
// Demonstruje przekierowania standardowego wejścia-wyjścia.
import java.io.*;

public class Redirecting {
  public static void main(String[] args)
```

[4] W rozdziale „Graficzne interfejsy użytkownika" pokazane jest jeszcze wygodniejsze rozwiązanie: program z interfejsem graficznym zawierający przewijane pole tekstowe.

```
  throws IOException {
    PrintStream console = System.out;
    BufferedInputStream in = new BufferedInputStream(
      new FileInputStream("Redirecting.java"));
    PrintStream out = new PrintStream(
      new BufferedOutputStream(
        new FileOutputStream("test.out")));
    System.setIn(in);
    System.setOut(out);
    System.setErr(out);
    BufferedReader br = new BufferedReader(
      new InputStreamReader(System.in));
    String s;
    while((s = br.readLine()) != null)
      System.out.println(s);
    out.close(); // Pamietaj o tym!
    System.setOut(console);
  }
} ///:~
```

Ten program przyłącza standardowe wejście do pliku oraz przekierowuje standardowe wyjście i standardowe wyjście błędu do innego pliku. Zwróć uwagę na to, że program zachowuje na początku referencję pierwotnego strumienia wyjścia `System.out`, aby przywrócić pierwotne ujście wyjścia na końcu programu.

Przekierowywanie wejścia-wyjścia działa na strumieniach bajtów, nie znaków, dlatego używamy klas `InputStream` i `OutputStream`, a nie `Reader` i `Writer`.

Sterowanie procesami zewnętrznymi

Często zachodzi potrzeba uruchomienia z wnętrza swojego programu innego programu systemu operacyjnego, a także zasilania tego programu danymi wejściowymi i przechwytywania jego wyjścia. Biblioteka Javy udostępnia w tym celu stosowne klasy.

Dość często uruchamia się program i przekazuje jego wyjście na konsolę; w tym podrozdziale przyjrzymy się narzędziu, które uprości to zadanie.

Przy okazji takiej operacji można się spodziewać błędów dwojakiego rodzaju: zwyczajnych błędów dających efekt w postaci wyjątków — które możemy wyrzucić jako wyjątki czasu wykonania — oraz błędów uruchomienia procesu zewnętrznego. Te ostatnie należałoby zgłaszać osobnym wyjątkiem:

```
//: net/mindview/util/OSExecuteException.java
package net.mindview.util;

public class OSExecuteException extends RuntimeException {
  public OSExecuteException(String why) { super(why); }
} ///:~
```

Aby uruchomić program, przekazujemy do metody `OSExecute.command()` taki ciąg polecenia, jaki wpisalibyśmy w wierszu poleceń w przypadku ręcznego uruchamiania. Polecenie

to jest przekazywane do konstruktora klasy `java.lang.ProcessBuilder` (spodziewającego się sekwencji obiektów `String`); powstaje obiekt `ProcessBuilder`:

```
//: net/mindview/util/OSExecute.java
// Uruchomienie polecenia zewnętrznego
// i przekazanie wyjścia na konsolę.
package net.mindview.util;
import java.io.*;

public class OSExecute {
  public static void command(String command) {
    boolean err = false;
    try {
      Process process =
        new ProcessBuilder(command.split(" ")).start();
      BufferedReader results = new BufferedReader(
        new InputStreamReader(process.getInputStream()));
      String s;
      while((s = results.readLine())!= null)
        System.out.println(s);
      BufferedReader errors = new BufferedReader(
        new InputStreamReader(process.getErrorStream()));
      // Błędy będą sygnalizowane wypisaniem komunikatów i zwróceniem
      // wartości niezerowej do procesu wywołującego:
      while((s = errors.readLine())!= null) {
        System.err.println(s);
        err = true;
      }
    } catch(Exception e) {
      // Dla Windows 2000, który wyrzuca wyjątek dla
      // domyślnego interpretera poleceń:
      if(!command.startsWith("CMD /C"))
        command("CMD /C " + command);
      else
        throw new RuntimeException(e);
    }
    if(err)
      throw new OSExecuteException("Błędy przy uruchamianiu " +
        command);
  }
} ///:~
```

Aby przechwycić standardowe wyjście uruchomionego programu zewnętrznego, należy wywołać metodę `getInputStream()`. Otrzymamy obiekt klasy `InputStream`, a więc coś, z czego będzie można odczytywać dane.

Wyniki są generowane przez program zewnętrzny wierszami, więc odczytujemy je wywołaniami metody `readLine()`. W powyższym przykładzie wyjście programu zewnętrznego jest najzwyczajniej wypisywane na konsolę, ale można je też przechwycić i przekazać jako wartość zwracaną metody `command()`.

Ewentualne komunikaty o błędach programu zewnętrznego trafiają do standardowego strumienia diagnostycznego i są stamtąd przechwytywane wywołaniem `getErrorStream()`. Przechwycone komunikaty są wypisywane, a dodatkowo zgłaszany jest wyjątek `OSExecuteException`, tak aby program wywołujący mógł jakoś obsłużyć problem.

Oto przykład użycia klasy OSExecute:

```
//: io/OSExecuteDemo.java
// Demonstruje przekierowanie standardowego wejścia-wyjścia.
import net.mindview.util.*;

public class OSExecuteDemo {
  public static void main(String[] args) {
    OSExecute.command("javap OSExecuteDemo");
  }
} /* Output:
Compiled from "OSExecuteDemo.java"
public class OSExecuteDemo extends java.lang.Object{
  public OSExecuteDemo();
  public static void main(java.lang.String[]);
}
*///:~
```

Mamy tu przykład wywołania zewnętrznego programu javap (wchodzącego w skład JDK) i przechwycenia wyniku zdekompilowania programu w języku Java.

Ćwiczenie 22. Zmień program *OSExecute.java* tak, aby zamiast wypisywać wyjście uruchamianego programu na standardowym wyjściu, zwracał wynik uruchomienia programu w postaci listy (`List`) ciągów `String`. Przetestuj nową wersję narzędzia (5).

Nowe wejście-wyjście

Wprowadzenie JDK 1.4, w pakiecie `java.nio.*` „nowej" biblioteki klas wejścia-wyjścia miało tylko jeden cel: szybkość. W rzeczywistości „stary" pakiet wejścia-wyjścia został ponownie zaimplementowany przy użyciu *nio*, aby skorzystać z jego szybkości. Dzięki temu korzyści uzyskiwane są nawet w przypadku, gdy klasy *nio* nie są bezpośrednio wykorzystywane. Wzrost prędkości działania można odnotować zarówno w operacjach wejścia-wyjścia związanych z obsługą plików (opisanych w niniejszym rozdziale), jak i operacjach sieciowych (opisanych w książce *Thinking in Enterprise Java*).

Szybkość uzyskiwana jest dzięki zastosowaniu struktur, które dokładniej odpowiadają sposobowi obsługi wejścia-wyjścia przez system operacyjny, czyli: *kanałów* i *buforów*. Można je sobie wyobrazić na przykładzie kopalni węgla kamiennego; kanał jest kopalnią posiadającą złoża węgla (danych), a bufor — wózkiem wysyłanym do szybu kopalni po węgiel. Wózek wraca z kopali wypełniony urobkiem, który jest z niego pobierany. Dzięki temu nie operujemy bezpośrednio na kanale; wszelkie operacje są wykonywane na buforze, który „odbiera" dane z kanału. Z kolei kanał bądź to pobiera dane z bufora, bądź też zapisuje je w nim.

Jedynym typem bufora, który bezpośrednio komunikuje się z kanałem, jest `ByteBuffer` — czyli bufor przechowujący nieprzetworzone bajty. Przeglądając w dokumentacji JDK informacje na temat tej klasy, można zauważyć, że jest ona stosunkowo prosta. Tworząc obiekt tej klasy, należy określić wielkość przydzielanej pamięci, a udostępniane przez nią metody pozwalają na pobieranie i zapis danych, zarówno w formie danych typów podstawowych, jak i nieprzetworzonych bajtów. Obiekty tej klasy nie pozwalają

jednak ani na pobieranie, ani na zapis obiektów, a nawet ciągów znaków. Jest to zatem klasa raczej niskiego poziomu i właśnie taką ma być, gdyż dzięki temu można ją najbardziej efektywnie implementować w większości systemów operacyjnych.

Aby powstała klasa `FileChannel`, w „starej" bibliotece wejścia-wyjścia zostały zmodyfikowane trzy klasy: `FileInputStream`, `FileOutputStream` oraz `RandomAccessFile` (umożliwiająca jednoczesny odczyt i zapis). Należy zwrócić uwagę, że są to klasy operujące na bajtach, co jest zgodne z niskim poziomem operacji wykonywanych przez bibliotekę *nio*. Na bazie klas znakowych — `Reader` oraz `Writer` — nie można tworzyć kanałów; niemniej jednak klasa `java.nio.channels.Channels` udostępnia metody pomocnicze umożliwiające przekształcanie kanałów na klasy `Reader` i `Writer`.

Oto prosty przykład przedstawiający, w jaki sposób, na bazie trzech typów strumieni, uzyskać różne rodzaje kanałów — do zapisu, do zapisu i odczytu oraz do odczytu:

```
//: io/GetChannel.java
// Wyłuskiwanie kanałów ze strumieni
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class GetChannel {
  private static final int BSIZE = 1024;
  public static void main(String[] args) throws Exception {
    // Zapis do pliku:
    FileChannel fc =
      new FileOutputStream("data.txt").getChannel();
    fc.write(ByteBuffer.wrap("Porcja danych ".getBytes()));
    fc.close();
    // Dodanie danych na koniec pliku:
    fc =
      new RandomAccessFile("data.txt", "rw").getChannel();
    fc.position(fc.size()); // Przesunięcie na koniec pliku
    fc.write(ByteBuffer.wrap("Jeszcze odrobina".getBytes()));
    fc.close();
    // Wczytanie pliku:
    fc = new FileInputStream("data.txt").getChannel();
    ByteBuffer buff = ByteBuffer.allocate(BSIZE);
    fc.read(buff);
    buff.flip();
    while(buff.hasRemaining())
      System.out.print((char)buff.get());
  }
} /* Output:
Porcja danych Jeszcze odrobina
*///:~
```

Każdy ze strumieni przedstawionych w powyższym programie pozwala na wywołanie metody `getChannel()`, zwracającej obiekt `FileChannel`. Uzyskiwany w ten sposób kanał jest stosunkowo prosty: można do niego przekazać `ByteBuffer` w celu zapisu lub odczytu oraz zablokować fragment pliku do wyłącznego dostępu (zagadnienie to zostanie dokładniej opisane w dalszej części rozdziału).

Bajty można umieszczać w obiekcie `ByteBuffer` bezpośrednio — przy użyciu jednej z metod „put", które zapisują jeden lub kilka bajtów bądź też wartości typów podstawowych.

Istnieje jednak również możliwość umieszczenia istniejącej tablicy bajtów w buforze ByteBuffer; służy do tego metoda wrap(), która także została użyta w przedstawionym przykładzie. W takim przypadku przekazana tablica nie jest kopiowana do bufora, lecz zamiast tego zostaje wykorzystana przez ByteBuffer jako miejsce przechowywania danych.

Plik *data.txt* jest ponownie otwierany przy użyciu klasy RandomAccessFile. Warto zwrócić uwagę, że FileChennel daje możliwość dowolnego poruszania się po pliku — w naszym przypadku program przechodzi na sam koniec jego zawartości, gdzie później zostanie dodany kolejny fragment tekstu.

W celu zapewnienia dostępu umożliwiającego jedynie odczyt danych należy utworzyć ByteBuffer posługując się statyczną metodą allocate(). Celem nowej biblioteki wejścia-wyjścia jest bardzo szybkie przenoszenie dużych ilości danych, dlatego też na potrzeby tworzonych obiektów ByteBuffer należy przydzielać znaczne obszary pamięci; w rzeczywistości 1 kilobajt przydzielany w przykładzie to zapewne znacznie mniej niż zazwyczaj będziesz chciał przydzielać (pisząc aplikację, trzeba będzie przeprowadzić eksperymenty, by określić optymalną ilość pamięci, jaką należy przydzielić).

Istnieje jednak możliwość jeszcze większego przyspieszenia programu. W tym celu zamiast metody allocate() należy wykorzystać allocateDirect(), tworzącą „bezpośredni" bufor, który w jeszcze większym stopniu jest skojarzony z systemem operacyjnym. Tworzenie buforów tego typu wiąże się jednak z większymi narzutami czasowymi, a poza tym implementacje tej klasy są różne w różnych systemach operacyjnych; dlatego też tworząc aplikację, należy eksperymentalnie sprawdzić, czy tworzenie bezpośrednich buforów w jakikolwiek sposób poprawia efektywność jej działania.

Aby móc odczytać dane z bufora po uprzednim zapisaniu ich przy użyciu metody FileChannel.read(), należy wywołać metodę flip() (tak, być może nie jest to eleganckie, ale należy pamiętać, że jest to klasa niskiego poziomu, stworzona w celu uzyskania jak najlepszej efektywności działania). Co więcej, aby wykorzystać ten sam bufor przy kolejnych operacjach odczytu, przed każdym wywołaniem metody read() należy wywoływać metodę clear(). Operacje te zostały przedstawione w prostym programie kopiującym zawartość plików, przedstawionym poniżej:

```
//: io/ChannelCopy.java
// Kopiowanie pliku za pomocą kanałów i buforów
// {Args: ChannelCopy.java test.txt}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class ChannelCopy {
  private static final int BSIZE = 1024;
  public static void main(String[] args) throws Exception {
    if(args.length != 2) {
      System.out.println("argumenty: plik-źródłowy plik-docelowy");
      System.exit(1);
    }
    FileChannel
      in = new FileInputStream(args[0]).getChannel(),
      out = new FileOutputStream(args[1]).getChannel();
    ByteBuffer buffer = ByteBuffer.allocate(BSIZE);
```

```
    while(in.read(buffer) != -1) {
      buffer.flip(); // Przygotowanie do zapisu
      out.write(buffer);
      buffer.clear(); // Przygotowanie do odczytu
    }
  }
} ///:~
```

Jak widać, jeden obiekt `FileChannel` jest otwierany do odczytu, a drugi do zapisu. Następnie tworzony jest obiekt `ByteBuffer`, a zwrócenie przez metodę `FileChannel.read()` wartości `-1` (co bez wątpienia jest pozostałością po systemach Unix i języku C) będzie oznaczać, że cała zawartość pliku wejściowego została odczytana. Po każdym wywołaniu metody `read()` zapisującej dane w buforze wywołanie `flip()` przygotowuje bufor i umożliwia pobranie jego zawartości przy użyciu metody `write()`. Po każdym odczycie informacje wciąż pozostają w buforze, dlatego też należy wywołać metodę `clear()`, która przypisuje domyślne wartości wewnętrznym wskaźnikom bufora, dzięki czemu jest on gotowy do zapisania kolejnej porcji danych.

Powyższy przykład nie jest jednak optymalnym sposobem wykonywania czynności tego typu. Istnieją specjalne metody — `transferTo()` oraz `transferFrom()` — pozwalające na bezpośrednie skojarzenie dwóch kanałów:

```
// : io/TransferTo.java
// Użycie metody transferTo() pomiędzy kanałami
// {Args: TransferTo.java TransferTo.txt}
import java.nio.channels.*;
import java.io.*;

public class TransferTo {
  public static void main(String[] args) throws Exception {
    if(args.length != 2) {
      System.out.println("argumenty: plik-źródłowy plik-docelowy");
      System.exit(1);
    }
    FileChannel
      in = new FileInputStream(args[0]).getChannel(),
      out = new FileOutputStream(args[1]).getChannel();
    in.transferTo(0, in.size(), out);
    // Albo:
    // out.transferFrom(in, 0, in.size());
  }
} ///:~
```

Operacje tego typu nie są zbyt częste, lecz warto wiedzieć, w jaki sposób można je zrealizować.

Konwersja danych

Przeglądając kod programu *GetChannel.java*, można zauważyć, że w celu wyświetlenia informacji z pliku kolejne bajty są pobierane i rzutowane do typu `char`. Rozwiązanie to jest nieco prymitywne — analizując klasę `java.nio.CharBuffer`, można zauważyć, że posiada ona metodę `toString()`, która zwraca ciąg znaków składający się ze znaków przechowywanych w danym buforze. Dzięki metodzie `asCharBuffer()` obiekt `ByteBuffer`

można przeglądać jako CharBuffer, dlaczego nie mielibyśmy więc z tego skorzystać? Rozwiązanie to jednak nie działa, o czym można się przekonać na podstawie pierwszego wiersza instrukcji wyjścia w poniższym programie:

```
//: io/BufferToText.java
// Konwersja tekstu do i z typu ByteBuffer
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.io.*;

public class BufferToText {
  private static final int BSIZE = 1024;
  public static void main(String[] args) throws Exception {
    FileChannel fc =
      new FileOutputStream("data2.txt").getChannel();
    fc.write(ByteBuffer.wrap("Jakiś tekst".getBytes()));
    fc.close();
    fc = new FileInputStream("data2.txt").getChannel();
    ByteBuffer buff = ByteBuffer.allocate(BSIZE);
    fc.read(buff);
    buff.flip();
    // Nie działa:
    System.out.println(buff.asCharBuffer());
    // Dekodowanie przy użyciu domyślnego zestawu
    // znaków danego systemu operacyjnego:
    buff.rewind();
    String encoding = System.getProperty("file.encoding");
    System.out.println("Dekodowane jako " + encoding + ": "
      + Charset.forName(encoding).decode(buff));
    // Moglibyśmy też zakodować coś, co da się wypisać:
    fc = new FileOutputStream("data2.txt").getChannel();
    fc.write(ByteBuffer.wrap(
      "Jakiś tekst".getBytes("UTF-16BE")));
    fc.close();
    // I ponownie spróbować odczytu:
    fc = new FileInputStream("data2.txt").getChannel();
    buff.clear();
    fc.read(buff);
    buff.flip();
    System.out.println(buff.asCharBuffer());
    // Użycie bufora CharBuffer przy zapisie:
    fc = new FileOutputStream("data2.txt").getChannel();
    buff = ByteBuffer.allocate(24); // Więcej niż trzeba
    buff.asCharBuffer().put("Jakiś tekst");
    fc.write(buff);
    fc.close();
    // Wczytanie i wypisanie:
    fc = new FileInputStream("data2.txt").getChannel();
    buff.clear();
    fc.read(buff);
    buff.flip();
    System.out.println(buff.asCharBuffer());
  }
} /* Output:
?????
Dekodowane jako Cp1250: Jakiś tekst
Jakiś tekst
Jakiś tekst
*///:~
```

Bufor zawiera zwyczajne bajty, a zatem, aby zamienić je na znaki, należy bądź to *zakodować* je w trakcie zapisywania w buforze (tak by podczas odczytu były poprawnymi znakami), bądź też *zdekodować* podczas pobierania z bufora. Operacje te można wykonać przy użyciu klasy `java.nio.charset.Charset`, udostępniającej narzędzia umożliwiające kodowanie danych w wielu różnych typach zbiorów znaków:

```
//: io/AvailableCharSets.java
// Wypisuje zestawy znaków i ich aliasy
import java.nio.charset.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class AvailableCharSets {
  public static void main(String[] args) {
    SortedMap<String,Charset> charSets =
      Charset.availableCharsets();
    Iterator<String> it = charSets.keySet().iterator();
    while(it.hasNext()) {
      String csName = it.next();
      printnb(csName);
      Iterator aliases =
        charSets.get(csName).aliases().iterator();
      if(aliases.hasNext())
        printnb(": ");
      while(aliases.hasNext()) {
        printnb(aliases.next());
        if(aliases.hasNext())
          printnb(", ");
      }
      print();
    }
  }
} /* Output:
Big5: csBig5
Big5-HKSCS: big5-hkscs, big5hk, big5-hkscs:unicode3.0, big5hkscs, Big5_HKSCS
EUC-JP: eucjis, x-eucjp, csEUCPkdFmtjapanese, eucjp,
Extended_UNIX_Code_Packed_Format_for_Japanese, x-euc-jp, euc_jp
EUC-KR: ksc5601, 5601, ksc5601_1987, ksc_5601, ksc5601-1987, euc_kr, ks_c_5601-1987, euckr,
csEUCKR
GB18030: gb18030-2000
GB2312: gb2312-1980, gb2312, EUC_CN, gb2312-80, euc-cn, euccn, x-EUC-CN
GBK: windows-936, CP936
...
*///:~
```

Wróćmy do przykładu *BufferToText.java*. Jeśli wywołamy metodę `rewind()` (aby przejść na sam początek danych przechowywanych w buforze) i użyjemy domyślnego systemowego zbioru znaków, aby zdekodować dane (wywołując metodę `decode()`), to uzyskaną w ten sposób zawartość bufora `CharBuffer` bez problemów będzie można wyświetlić na konsoli. Domyślny zbiór znaków można określić za pomocą wywołania `System.getProperty ("file.encoding")`, zwracającego ciąg znaków zawierający nazwę zbioru znaków. Po przekazaniu tej nazwy w wywołaniu metody `Charset.forName()`, pobierany jest obiekt `Charset`, który następnie można wykorzystać do zdekodowania ciągu znaków.

Alternatywnym rozwiązaniem jest wywołanie metody encode() i przekazanie do niej wybranego zbioru znaków. W ten sposób, już podczas pobierania z pliku, dane zostaną przekształcone do postaci umożliwiającej ich wyświetlenie, o czym można się przekonać w trzeciej części programu *BufferToText.java*. W przykładzie tym dane zakodowane w UTF-16BE są zapisywane w pliku tekstowym, następnie, aby je odczytać, wystarczy je skonwertować i umieścić w obiekcie CharBuffer, a uzyskany w ten sposób tekst będzie zgodny z oczekiwaniami.

Ostatnia część programu pokazuje, co się dzieje w przypadku zapisywania danych w buforze ByteBuffer poprzez CharBuffer (w dalszej części rozdziału podam więcej informacji na ten temat). Należy zauważyć, że na potrzeby bufora zarezerwowano 24 bajty. Ponieważ każdy znak wymaga dwóch bajtów, w buforze tym można zapisać 12 znaków; jednak ciąg „Jakiś tekst" ma ich tylko 11. Jak pokazują wyniki programu, pozostałe — „zerowe" — bajty wciąż są widoczne po wyświetleniu zawartości bufora CharBuffer przy użyciu metody toString().

Ćwiczenie 23. Stwórz i przetestuj metodę pomocniczą prezentującą znaki przechowywane w buforze CharBuffer aż do miejsca, w którym kończą się znaki nadające się do wyświetlania (6).

Pobieranie podstawowych typów danych

Choć ByteBuffer jedynie przechowuje bajty, klasa ta definiuje metody umożliwiające zwracanie tych bajtów w formie danych dowolnych typów podstawowych. Poniższy przykład przedstawia wykorzystanie tych metod do zapisu i pobierania różnego typu danych:

```
//: io/GetData.java
// Wyłuskiwanie różnych reprezentacji z bufora ByteBuffer
import java.nio.*;
import static net.mindview.util.Print.*;

public class GetData {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(BSIZE);
    // Przydział automatycznie zeruje bufor ByteBuffer:
    int i = 0;
    while(i++ < bb.limit())
      if(bb.get() != 0)
        print("niezerowa");
    print("i = " + i);
    bb.rewind();
    // Zapisanie i odczytanie tablicy znaków:
    bb.asCharBuffer().put("Jak leci?");
    char c;
    while((c = bb.getChar()) != 0)
      printnb(c + " ");
    print();
    bb.rewind();
    // Zapisanie i odczytanie wartości typu short:
    bb.asShortBuffer().put((short)471142);
    print(bb.getShort());
    bb.rewind();
```

```
    // Zapisanie i odczytanie wartości typu int:
    bb.asIntBuffer().put(99471142);
    print(bb.getInt());
    bb.rewind();
    // Zapisanie i odczytanie wartości typu long:
    bb.asLongBuffer().put(99471142);
    print(bb.getLong());
    bb.rewind();
    // Zapisanie i odczytanie wartości typu float:
    bb.asFloatBuffer().put(99471142);
    print(bb.getFloat());
    bb.rewind();
    // Zapisanie i odczytanie wartości typu double:
    bb.asDoubleBuffer().put(99471142);
    print(bb.getDouble());
    bb.rewind();
  }
} /* Output:
i = 1025
Jak leci?
12390
99471142
99471142
9.9471144E7
9.9471142E7
*///:~
```

Po przydzieleniu pamięci obiektowi `ByteBuffer` jego zawartość jest sprawdzana, aby się upewnić, czy jest ona automatycznie wyzerowana, co faktycznie ma miejsce. Sprawdzane są wszystkie 1024 wartości (aż do granicy wyznaczonej metodą `limit()`) i wszystkie mają wartość równą 0.

Najprostszym sposobem wstawienia wartości typu podstawowego do bufora `ByteBuffer` jest pobranie odpowiedniego „widoku" tego bufora przy użyciu jednej z metod `asCharBuffer()`, `asShortBuffer()` itd. oraz wywołanie metody `put()` uzyskanego obiektu. Jak widać, ten sposób zapisu jest stosowany dla wszystkich typów podstawowych. Jedynie metoda `put()` dla bufora `ShortBuffer` jest nieco dziwna, gdyż wymaga zastosowania rzutowania (które, jak widać, przycina i zmienia wartość wynikową). Metody `put()` wszystkich pozostałych „widoków" buforów nie wymagają żadnego rzutowania.

Widoki buforów

„Widok bufora" umożliwia spojrzenie na `ByteBuffer` w taki sposób, jak gdyby zawierał on dane konkretnego, podstawowego typu. `ByteBuffer` wciąż przechowuje dane dla widoku, a zatem wszelkie modyfikacje wprowadzane w widoku powodują zmianę zawartości obiektu `ByteBuffer`. Dzięki takiemu rozwiązaniu można wygodnie zapisywać w buforze dane typów podstawowych, o czym można się było przekonać w poprzednim przykładzie. Widok pozwala także na odczytywanie z bufora danych konkretnego, podstawowego typu, i to zarówno pojedynczo (na co pozwala także `ByteBuffer`), jak i grupami (do tablic). Poniższy przykład manipuluje liczbami całkowitymi zapisanymi w buforze `ByteBuffer` za pośrednictwem obiektu `IntBuffer`:

```
// io/IntBufferDemo.java
// Manipulowanie wartościami typu int w buforze ByteBuffer
// za pośrednictwem widoku IntBuffer
import java.nio.*;

public class IntBufferDemo {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(BSIZE);
    IntBuffer ib = bb.asIntBuffer();
    // Zapisanie tablicy wartości typu int:
    ib.put(new int[]{ 11, 42, 47, 99, 143, 811, 1016 });
    // Odczyt i zapis spod i na konkretną pozycję:
    System.out.println(ib.get(3));
    ib.put(3, 1811);
    // Ustawienie nowego limitu przed przewinięciem bufora.
    ib.flip();
    while(ib.hasRemaining()) {
      int i = ib.get();
      System.out.println(i);
    }
  }
} /* Output:
99
11
42
47
1811
143
811
1016
*///:~
```

W pierwszej kolejności, przy użyciu przeciążonej metody `put()`, w buforze zapisywana jest tablica danych typu `int`. Kolejne wywołania metod `get()` oraz `put()` bezpośrednio operują na liczbach całkowitych przechowywanych w konkretnych miejscach bufora `ByteBuffer`. Należy zauważyć, że w przypadku operowania na podstawowych typach danych ten dostęp do danych przechowywanych w konkretnych miejscach bufora można także uzyskać, korzystając bezpośrednio z metod obiektu `ByteBuffer`.

Kiedy, przy wykorzystaniu odpowiedniego widoku, w obiekcie `ByteBuffer` zostaną już zapisane liczby `int` lub dowolnego innego, podstawowego typu, obiekt ten będzie można bezpośrednio zapisać w kanale. Równie łatwo można odczytać dane z kanału i użyć widoku bufora do skonwertowania ich do konkretnego, podstawowego typu danych. Poniższy przykład interpretuje tę samą sekwencję bajtów jako liczby typów `short`, `int`, `float`, `long` oraz `double`, wykorzystując w tym celu różne widoki tego samego obiektu `ByteBuffer`:

```
// io/ViewBuffers.java
import java.nio.*;
import static net.mindview.util.Print.*;

public class ViewBuffers {
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.wrap(
      new byte[]{ 0, 0, 0, 0, 0, 0, 0, 'a' });
    bb.rewind();
```

```
    printnb("Bufor danych typu byte ");
    while(bb.hasRemaining())
      printnb(bb.position()+ " -> " + bb.get() + ", ");
    print();
    CharBuffer cb =
      ((ByteBuffer)bb.rewind()).asCharBuffer();
    printnb("Bufor danych typu char ");
    while(cb.hasRemaining())
      printnb(cb.position() + " -> " + cb.get() + ", ");
    print();
    FloatBuffer fb =
      ((ByteBuffer)bb.rewind()).asFloatBuffer();
    printnb("Bufor danych typu float ");
    while(fb.hasRemaining())
      printnb(fb.position()+ " -> " + fb.get() + ", ");
    print();
    IntBuffer ib =
      ((ByteBuffer)bb.rewind()).asIntBuffer();
    printnb("Bufor danych typu int ");
    while(ib.hasRemaining())
      printnb(ib.position()+ " -> " + ib.get() + ", ");
    print();
    LongBuffer lb =
      ((ByteBuffer)bb.rewind()).asLongBuffer();
    printnb("Bufor danych typu long ");
    while(lb.hasRemaining())
      printnb(lb.position()+ " -> " + lb.get() + ", ");
    print();
    ShortBuffer sb =
      ((ByteBuffer)bb.rewind()).asShortBuffer();
    printnb("Bufor danych typu short ");
    while(sb.hasRemaining())
      printnb(sb.position()+ " -> " + sb.get() + ", ");
    print();
    DoubleBuffer db =
      ((ByteBuffer)bb.rewind()).asDoubleBuffer();
    printnb("Bufor danych typu double ");
    while(db.hasRemaining())
      printnb(db.position()+ " -> " + db.get() + ", ");
  }
} /* Output:
Bufor danych typu byte 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 0, 4 -> 0, 5 -> 0, 6 -> 0, 7 -> 97,
Bufor danych typu char 0 -> , 1 -> , 2 -> , 3 -> a,
Bufor danych typu float 0 -> 0.0, 1 -> 1.36E-43,
Bufor danych typu int 0 -> 0, 1 -> 97,
Bufor danych typu long 0 -> 97,
Bufor danych typu short 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 97,
Bufor danych typu double 0 -> 4.8E-322,
*///:~
```

Obiekt `ByteBuffer` jest tworzony poprzez „opakowanie" tablicy zawierającej osiem bajtów; przy użyciu różnych widoków bufora jej zawartość jest następnie wyświetlana w formie danych wszystkich dostępnych typów podstawowych. Poniższy diagram pokazuje, w jaki sposób, przy użyciu różnych typów buforów, te same dane są odmiennie interpretowane:

0	0	0	0	0	0	0	97	byte
						a		char
0		0		0		97		short
0				97				int
0.0				1.36E-43				float
97								long
4.8E-322								double

Powyższy diagram odpowiada wynikom generowanym przez program.

Ćwiczenie 24. Zmodyfikuj program *IntBufferDemo.java* tak, aby używał typu `double` (1).

Kolejność zapisu bajtów

Na różnych komputerach mogą być wykorzystywane różne sposoby rozmieszczania bajtów w zapisywanych danych. W kolejności „big endian" (wykorzystywanej przez procesory PowerPC, Motorola, SPARC) najbardziej znaczący bajt jest umieszczany w komórce pamięci o najniższym adresie, natomiast w kolejności „little endian" (wykorzystywanej przez procesory Intel, AMD) — w komórce pamięci o najwyższym adresie. Może się zdarzyć, że podczas zapisywania danych, których wielkość przekracza jeden bajt, takich jak `int`, `float` itd., trzeba będzie uwzględniać kolejność rozmieszczania bajtów. Klasa `ByteBuffer` przechowuje dane, zapisując je w kolejności „big endian"; w takim samym porządku dane są przesyłane przez Internet. Kolejność zapisu bajtów w buforze `ByteBuffer` można zmienić za pomocą metody `order()`, przekazując jako argument jej wywołania jedną z wartości `ByteOrder.BIG_ENDIAN` lub `ByteOrder.LITTLE_ENDIAN`.

Przeanalizujmy przykład bufora zawierającego dwa bajty:

0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1
b1								b2							

Odczytując zawartość takiego bufora jako liczbę typu `short` (`ByteBuffer.asShortBuffer()`), uzyskamy wartość 97 (00000000 01100001); jednak po zmianie kolejności na „little endian" uzyskamy wartość 24832 (01100001 00000000).

Poniższy przykład pokazuje skutki modyfikacji kolejności zapisu bajtów tworzących znaki:

```
//: io/Endians.java
// Kolejność zapisu bajtów a składowanie danych.
import java.nio.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Endians {
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.wrap(new byte[12]);
    bb.asCharBuffer().put("abcdef");
```

```
    print(Arrays.toString(bb.array()));
    bb.rewind();
    bb.order(ByteOrder.BIG_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    print(Arrays.toString(bb.array()));
    bb.rewind();
    bb.order(ByteOrder.LITTLE_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    print(Arrays.toString(bb.array()));
  }
} /* Output:
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102, 0]
*///:~
```

Obszar przydzielony buforowi `ByteBuffor` jest na tyle duży, że w formie zewnętrznego bufora można w nim zapisać wszystkie bajty tablicy znaków; dzięki temu będzie je można wyświetlić przy użyciu metody `array()`. Metoda ta jest „opcjonalna" i można ją wywoływać wyłącznie w przypadkach, gdy bufor został utworzony na bazie tablicy; w przeciwnym razie zostanie zgłoszony wyjątek `UnsupportedOperationException`.

Tablica znaków jest zapisywana w buforze dzięki wykorzystaniu widoku `CharBuffer`. Prezentacja bajtów stanowiących zawartość bufora pokazuje, że domyślna kolejność rozmieszczania bajtów jest taka sama, jak jawnie zażądana kolejność „big endian". Widać również, że w przypadku użycia kolejności „little endian" kolejność bajtów jest zamieniana.

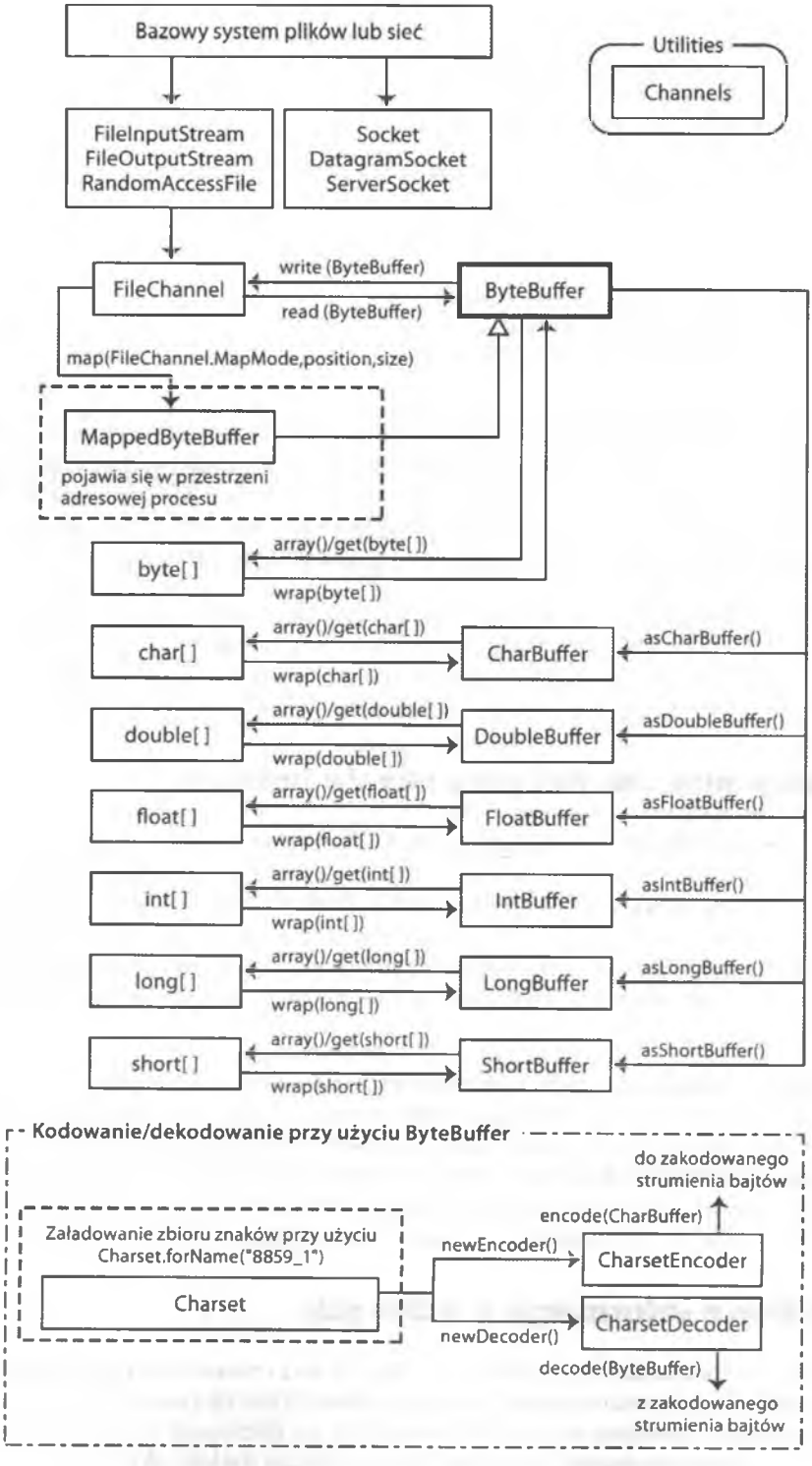
Manipulowanie danymi przy użyciu buforów

Diagram przedstawiony na następnej stronie pokazuje relację pomiędzy klasami `nio`, na jego podstawie można zorientować się, jakie są możliwości przekazywania i konwersji danych. Na przykład, aby zapisać w pliku tablicę bajtów, wystarczy utworzyć na jej podstawie bufor (przy użyciu metody `ByteBuffer.wrap()`), następnie, posługując się metodą `get-Channel()`, otworzyć kanał operujący na strumieniu `FileOutputStream`, korzystając z niego, utworzyć `FileChannel`, w którym można już zapisać dane z obiektu `ByteBuffer`.

Należy zauważyć, że zapis i odczyt danych z kanałów jest możliwy wyłącznie za pośrednictwem obiektów `ByteBuffer` oraz że można tworzyć wyłącznie niezależne bufory typów podstawowych; istnieje także możliwość utworzenia takich buforów na podstawie bufora `ByteBuffer` przy użyciu odpowiednich metod „as". Nie są to jednak poważne ograniczenia, gdyż dane typów podstawowych można zapisywać i odczytywać w buforach `ByteBuffer` za pośrednictwem odpowiednich „widoków".

Szczegółowe informacje o buforach

Bufor — obiekt `Buffer` — składa się z danych oraz czterech indeksów umożliwiających efektywne operowanie na tych danych; indeksami tymi są: *znacznik*, *położenie*, *granica*, *pojemność*. Dostępne są metody pozwalające na odczytanie wartości tych indeksów, przywrócenie im wartości domyślnych oraz pobranie wartości aktualnej.


Bazowy system plików lub sieć
Utilities
Channels
FileInputStream
FileOutputStream
RandomAccessFile
Socket
DatagramSocket
ServerSocket
FileChannel
write (ByteBuffer)
read (ByteBuffer)
ByteBuffer
map(FileChannel.MapMode,position,size)
MappedByteBuffer
pojawia się w przestrzeni adresowej procesu
byte[ ]
array()/get(byte[ ])
wrap(byte[ ])
char[ ]
array()/get(char[ ])
wrap(char[ ])
CharBuffer
asCharBuffer()
double[ ]
array()/get(double[ ])
wrap(double[ ])
DoubleBuffer
asDoubleBuffer()
float[ ]
array()/get(float[ ])
wrap(float[ ])
FloatBuffer
asFloatBuffer()
int[ ]
array()/get(int[ ])
wrap(int[ ])
IntBuffer
asIntBuffer()
long[ ]
array()/get(long[ ])
wrap(long[ ])
LongBuffer
asLongBuffer()
short[ ]
array()/get(short[ ])
wrap(short[ ])
ShortBuffer
asShortBuffer()
Kodowanie/dekodowanie przy użyciu ByteBuffer
do zakodowanego strumienia bajtów
encode(CharBuffer)
Załadowanie zbioru znaków przy użyciu Charset.forName("8859_1")
newEncoder()
CharsetEncoder
Charset
CharsetDecoder
newDecoder()
decode(ByteBuffer)
z zakodowanego strumienia bajtów

Metoda	Działanie
`capacity()`	Zwraca wartość indeksu *pojemność* bufora.
`clear()`	Czyści bufor; przypisuje indeksowi *położenie* wartość 0, a indeksowi *granica* wartość indeksu *pojemność*. Metoda ta jest wywoływana w celu usunięcia bieżącej zawartości bufora.
`flip()`	Przypisuje indeksowi *granica* wartość indeksu *położenie*, a indeksowi *położenie* wartość 0. Metoda ta przygotowuje bufor od odczytu po wcześniejszym zapisaniu w nim informacji.
`limit()`	Zwraca wartość indeksu *granica*.
`limit(int lim)`	Ustawia wartość indeksu *granica*.
`mark()`	Ustawia wartość indeksu *znacznik*, przypisując mu wartość indeksu *położenie*.
`position()`	Zwraca wartość indeksu *położenie*.
`position(int pos)`	Ustawia wartość indeksu *położenie*.
`remaining()`	Zwraca wartość indeksu *granica* pomniejszoną o wartość indeksu *położenie*.
`hasRemaining()`	Zwraca wartość `true`, jeśli pomiędzy miejscami wyznaczanymi położeniem indeksów *położenie* oraz *granica* są jakieś dane.

Metody wstawiające i pobierające dane z buforów odpowiednio aktualizują powyższe indeksy.

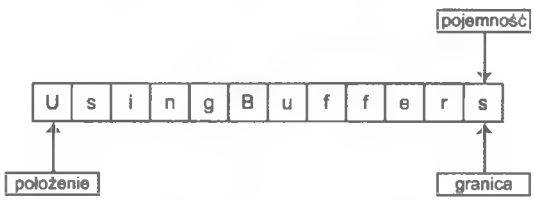
Poniższy przykład miesza kolejność zapisu znaków w buforze `CharBuffer`, wykorzystując w tym celu bardzo prosty algorytm (zamienia kolejność znaków sąsiadujących ze sobą):

```
//: io/UsingBuffers.java
import java.nio.*;
import static net.mindview.util.Print.*;

public class UsingBuffers {
  private static void symmetricScramble(CharBuffer buffer){
    while(buffer.hasRemaining()) {
      buffer.mark();
      char c1 = buffer.get();
      char c2 = buffer.get();
      buffer.reset();
      buffer.put(c2).put(c1);
    }
  }
  public static void main(String[] args) {
    char[] data = "UsingBuffers".toCharArray();
    ByteBuffer bb = ByteBuffer.allocate(data.length * 2);
    CharBuffer cb = bb.asCharBuffer();
    cb.put(data);
    print(cb.rewind());
    symmetricScramble(cb);
    print(cb.rewind());
    symmetricScramble(cb);
    print(cb.rewind());
  }
} /* Output:
UsingBuffers
sUniBgfuefsr
UsingBuffers
*///:~
```

Bufor `CharBuffer` można by utworzyć bezpośrednio, wykorzystując w tym celu tablicę znaków i metodę `wrap()`; niemniej jednak w powyższym przykładzie tworzony jest obiekt `ByteBuffer`, a obiekt `CharBuffer` jest tworzony jako jego „widok". Rozwiązanie to ma podkreślić fakt, że zawsze naszym celem jest operowanie buforem `ByteBuffer`, gdyż to właśnie obiekty tej klasy pośredniczą w wymianie danych z kanałami.

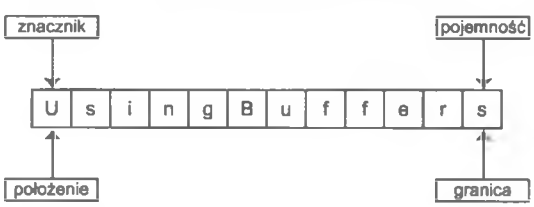
Oto jak wygląda zawartość bufora na wejściu do metody `symmetricScramble()`:



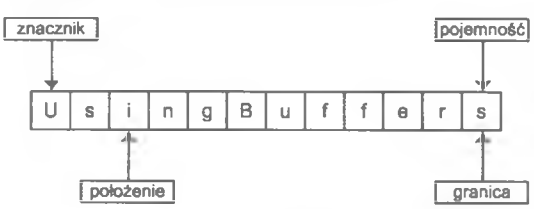
Indeks *położenie* wskazuje na pierwszy element bufora, a indeksy *pojemność* i *granica* na element ostatni.

W metodzie `symmetricScramble()` pętla `while` jest wykonywana aż do momentu zrównania wartości indeksów *położenie* i *granica*. Wartość znacznika *położenie* zmienia się, gdy wywoływane są względne metody `get()` oraz `put()`. Można także korzystać z bezwzględnych metod `get()` oraz `put()`, wymagających podania indeksu określającego miejsce, w którym zostanie wykonana operacja odczytu bądź zapisu. Metody te nie modyfikują wartości indeksu *położenie*.

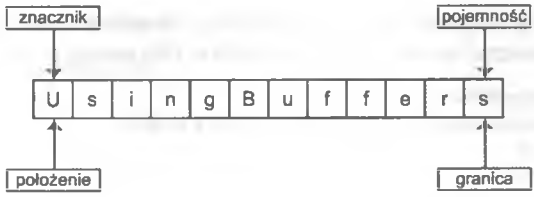
Gdy sterowanie wchodzi od pętli `while` wartość indeksu *znacznik* jest ustawiana przy użyciu metody `mark()`. Poniższy diagram przedstawia stan bufora po tym wywołaniu:



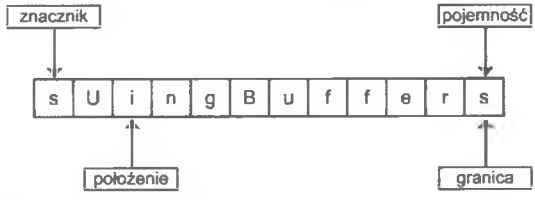
Dwa wywołania względnej metody `get()` zapisują pierwsze dwa znaki bufora w zmiennych `c1` i `c2`. Oto stan bufora po tych wywołaniach:



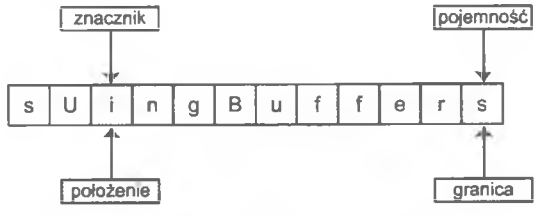
Aby dokonać zamiany znaków, należy zapisać wartość zmiennej `c2` w miejscu, dla którego indeks *położenie* ma wartość `0`, a zmienną `c1` w miejscu, dla którego znacznik ten ma wartość `1`. Można to zrobić na dwa sposoby — używając bezwzględnych metod `put()` bądź też przypisując indeksowi *położenie* wartość indeksu *znacznik*, co robi metoda `reset()`:



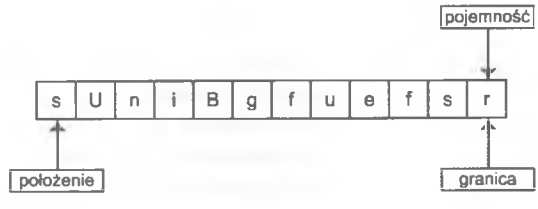
Dwa wywołania metody `put()` zapisują w buforze wartości zmiennych `c2` oraz `c1`:



Podczas kolejnej iteracji pętli indeksowi *znacznik* przypisywana jest wartość indeksu *położenie*:



Powyższy proces jest kontynuowany aż do momentu przetworzenia całego bufora. Na samym końcu działania pętli `while` indeks *położenie* wskazuje na koniec bufora. Podczas wyświetlania zawartości bufora prezentowane są wyłącznie znaki pomiędzy miejscami wskazywanymi przez indeksy *położenie* i *granica*. A zatem, aby wyświetlić całą zawartość bufora, należy przenieść indeks *położenie* na sam początek bufora, co można zrobić, używając metody `rewind()`. Oto stan bufora po wywołaniu metody `rewind()` (która jednocześnie sprawia, że wartość indeksu *znacznik* staje się niezdefiniowana):



Ponowne wywołanie metody `symmetricScramble()` powoduje ponowne wykonanie tych samych operacji na buforze, a tym samym — przywrócenie go do oryginalnego stanu.

Pliki odwzorowywane w pamięci

Możliwość odwzorowywania plików w pamięci pozwala na tworzenie i modyfikowanie plików, które są zbyt duże, aby je w całości wczytać do pamięci. Dzięki plikom odwzorowywanym w pamięci można udawać, że cały plik znajduje się w pamięci oraz można go

traktować, jak gdyby był bardzo dużą tablicą. Możliwość taka znacznie upraszcza kod, jaki należy stworzyć w celu modyfikacji pliku. Oto prosty przykład:

```
//: io/LargeMappedFiles.java
// Tworzenie bardzo dużego pliku z odwzorowaniem w pamięci.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
import static net.mindview.util.Print.*;

public class LargeMappedFiles {
  static int length = 0x8FFFFFF; // 128 MB
  public static void main(String[] args) throws Exception {
    MappedByteBuffer out =
      new RandomAccessFile("test.dat", "rw").getChannel()
      .map(FileChannel.MapMode.READ_WRITE, 0, length);
    for(int i = 0; i < length; i++)
      out.put((byte)'x');
    print("Zapis zakończony");
    for(int i = length/2; i < length/2 + 6; i++)
      printnb((char)out.get(i));
  }
} ///:~
```

Aby móc zarówno zapisywać, jak i odczytywać zawartość pliku, należy utworzyć obiekt `RandomAccessFile`, pobrać kanał do pliku, a następnie wywołać metodę `map()` w celu utworzenia obiektu `MappedByteBuffer`, będącego szczególnym rodzajem bezpośredniego bufora. Należy zauważyć, że trzeba podać punkt początkowy oraz wielkość obszaru, jaki będzie odwzorowywany w pliku; oznacza to, że istnieje możliwość odwzorowywania niewielkich obszarów dużych plików.

`MappedByteBuffer` dziedziczy po klasie `ByteBuffer`, a zatem udostępnia wszystkie metody tej klasy. W powyższym przykładzie przedstawione zostały wyłącznie bardzo proste metody `put()` oraz `get()`, niemniej jednak można także stosować metodę `asCharBuffer()` oraz wszystkie inne.

Plik utworzony przez program przedstawiony w ostatnim przykładzie ma 128 MB wielkości, co zapewne przekracza obszar dozwolony przez system operacyjny w jednym przydziale. Wydaje się, że od razu dostępna jest cała zawartość pliku, gdyż jedynie fragmenty pliku są pobierane do pamięci, a następnie usuwane z niej i zastępowane innymi. W ten sposób, bez większych problemów, można modyfikować bardzo duże pliki (do 2 GB). Warto zauważyć, że w celu uzyskania jak największej efektywności działania w maksymalnym stopniu są wykorzystywane narzędzia odwzorowywania plików, jakie daje system operacyjny.

Efektywność

Choć efektywność „starej" biblioteki wejścia-wyjścia bazującej na strumieniach została poprawiona poprzez zaimplementowanie jej przy użyciu `nio`, wykorzystanie plików odwzorowywanych w pamięci wiąże się zazwyczaj z ogromnym jej wzrostem. Poniższy program stanowi proste porównanie efektywności:

```
// : io/MappedIO.java
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class MappedIO {
  private static int numOfInts = 4000000;
  private static int numOfUbuffInts = 200000;
  private abstract static class Tester {
    private String name;
    public Tester(String name) { this.name = name; }
    public void runTest() {
      System.out.print(name + ": ");
      try {
        long start = System.nanoTime();
        test();
        double duration = System.nanoTime() - start;
        System.out.format("%.2f\n", duration/1.0e9);
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
    public abstract void test() throws IOException;
  }
  private static Tester[] tests = {
    new Tester("Zapis w strumieniu") {
      public void test() throws IOException {
        DataOutputStream dos = new DataOutputStream(
          new BufferedOutputStream(
            new FileOutputStream(new File("temp.tmp"))));
        for(int i = 0; i < numOfInts; i++)
          dos.writeInt(i);
        dos.close();
      }
    },
    new Tester("Zapis w pliku odwzorowanym w pamięci") {
      public void test() throws IOException {
        FileChannel fc =
          new RandomAccessFile("temp.tmp", "rw")
          .getChannel();
        IntBuffer ib = fc.map(
          FileChannel.MapMode.READ_WRITE, 0, fc.size())
          .asIntBuffer();
        for(int i = 0; i < numOfInts; i++)
          ib.put(i);
        fc.close();
      }
    },
    new Tester("Odczyt ze strumienia") {
      public void test() throws IOException {
        DataInputStream dis = new DataInputStream(
          new BufferedInputStream(
            new FileInputStream("temp.tmp")));
        for(int i = 0; i < numOfInts; i++)
          dis.readInt();
        dis.close();
      }
    },
```

```
    new Tester("Odczyt z pliku odwzorowanego w pamięci") {
      public void test() throws IOException {
        FileChannel fc = new FileInputStream(
          new File("temp.tmp")).getChannel();
        IntBuffer ib = fc.map(
          FileChannel.MapMode.READ_ONLY, 0, fc.size())
          .asIntBuffer();
        while(ib.hasRemaining())
          ib.get();
        fc.close();
      }
    },
    new Tester("Odczyt-zapis w strumieniu") {
      public void test() throws IOException {
        RandomAccessFile raf = new RandomAccessFile(
          new File("temp.tmp"), "rw");
        raf.writeInt(1);
        for(int i = 0; i < numOfUbuffInts; i++) {
          raf.seek(raf.length() - 4);
          raf.writeInt(raf.readInt());
        }
        raf.close();
      }
    },
    new Tester("Odczyt-zapis w pliku odwzorowanym w pamięci") {
      public void test() throws IOException {
        FileChannel fc = new RandomAccessFile(
          new File("temp.tmp"), "rw").getChannel();
        IntBuffer ib = fc.map(
          FileChannel.MapMode.READ_WRITE, 0, fc.size())
          .asIntBuffer();
        ib.put(0);
        for(int i = 1; i < numOfUbuffInts; i++)
          ib.put(ib.get(i - 1));
        fc.close();
      }
    }
  };
  public static void main(String[] args) {
    for(Tester test : tests)
      test.runTest();
  }
} /* Output: (90% match)
Zapis w strumieniu: 0,56
Zapis w pliku odwzorowanym w pamięci: 0,12
Odczyt ze strumienia: 0,80
Odczyt z pliku odwzorowanego w pamięci: 0,07
Odczyt-zapis w strumieniu: 5,32
Odczyt-zapis w pliku odwzorowanym w pamięci: 0,02
*///:~
```

Przedstawiona już we wcześniejszej części książki metoda `runTest()` jest wykorzystywana przez *metodę-wzorzec* (z wzorca projektowego *Template Method*), stanowiącą szkielet testowy dla różnych implementacji metody `test()`, zdefiniowanych w anonimowych klasach wewnętrznych. Każda z tych podklas przeprowadza jeden, konkretny typ testu, a zatem metody `test()` udostępniają prototyp pozwalający na realizację różnych operacji wejścia-wyjścia.

Choć wydaje się, że zapis do pliku odwzorowanego w pamięci wykorzystuje obiekt `FileOutputStream`, wszystkie operacje zapisu muszą być wykonywane przy użyciu obiektu `RandomAccessFile`, tak jak operacje zapisu i odczytu przedstawione w powyższym przykładzie.

Należy pamiętać, że pomiar dokonywany przez metodę `test()` obejmuje także czas inicjalizacji wszelkich obiektów wejścia-wyjścia, a zatem, nawet jeśli uwzględnimy fakt, że przygotowanie obiektów koniecznych do obsługi plików odwzorowywanych w pamięci jest czasochłonne, to i tak w porównaniu ze strumieniami ogólny zysk jest bardzo duży.

Ćwiczenie 25. Spróbuj w przykładach z bieżącego rozdziału zmienić instrukcje `ByteBuffer.allocate()` na `ByteBuffer.allocateDirect()`. Pokaż różnice w osiąganej wydajności, ale zwróć też uwagę na zauważalne wydłużenie czasu uruchamiania programów (6).

Ćwiczenie 26. Zmodyfikuj program *JGrep.java*, aby korzystał z plików odwzorowywanych w pamięci dostępnych w bibliotece `nio` (3).

Blokowanie plików

Blokowanie plików pozwala na synchronizację dostępu do pliku jako zasobu wspólnego. Jednak dwa wątki rywalizujące o dostęp do pliku mogą być wykonywane przez różne wirtualne maszyny Javy bądź też jeden z nich może być rodzimym wątkiem systemu operacyjnego. Blokowanie plików jest jednak widoczne dla innych procesów systemowych, gdyż mechanizmy blokowania zaimplementowane w Javie są bezpośrednio odwzorowywane na mechanizmy blokowania systemu operacyjnego.

Oto prosty przykład blokowania:

```
//: io/FileLocking.java
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;

public class FileLocking {
  public static void main(String[] args) throws Exception {
    FileOutputStream fos= new FileOutputStream("file.txt");
    FileLock fl = fos.getChannel().tryLock();
    if(fl != null) {
      System.out.println("Plik zablokowany");
      TimeUnit.MILLISECONDS.sleep(100);
      fl.release();
      System.out.println("Blokada zwolniona");
    }
    fos.close();
  }
} /* Output:
Plik zablokowany
Blokada zwolniona
*///:~
```

Obiekt `FileLock`, reprezentujący blokadę dostępu do całego pliku, można uzyskać, wywołując metodę `tryLock()` lub `lock()` obiektu `FileChannel`. (Kanał `SocketChannel`, `DatagramChannel` oraz `ServerSocketChannel` nie potrzebują mechanizmu blokowania, gdyż

z założenia są wykorzystywane tylko przez jeden proces; nie tworzy się przecież wspólnych gniazd sieciowych łączących dwa procesy.) Metoda `tryLock()` nie wstrzymuje działania programu. Podejmuje ona próbę zablokowania dostępu do pliku, lecz jeśli nie jest to możliwe (czyli gdy inny proces wcześniej zablokował dostęp do tego pliku, a blokada nie jest współdzielona), jej działanie się kończy. Z kolei metoda `lock()` wstrzymuje realizację programu aż do momentu, gdy uda się zablokować dostęp do pliku, bądź do chwili, gdy wątek, który ją wywołał, zostanie przerwany, albo do momentu zamknięcia kanału użytego do wywołania metody. Blokada dostępu do pliku jest usuwana przez wywołanie metody `FileLock.release()`.

Istnieje także możliwość zablokowania dostępu do fragmentu pliku; w tym celu należy użyć wywołania:

```
tryLock(long położenie, long wielkość, boolean współdzielona)
```

lub

```
lock(long położenie, long wielkość, boolean współdzielona)
```

które blokuje dostęp do fragmentu pliku (`wielkość - położenie`). Ostatni argument określa, czy blokada dostępu do pliku jest współdzielona czy nie.

Blokady uzyskane przez dwie powyższe metody nie są w żaden sposób modyfikowane wraz ze zmianami wielkości pliku, co odróżnia je od blokad tworzonych przy użyciu metod bezargumentowych — `tryLock()` oraz `lock()`. Kiedy dostęp do obszaru `położenie` do `położenie+rozmiar` zostanie zablokowany, a zawartość pliku zostanie powiększona i przekroczy granicę `położenie+rozmiar`, nie będzie żadnych ograniczeń w dostępie do tej dodatkowej części pliku. Posługując się metodami bezargumentowymi, można zablokować dostęp do całego pliku, nawet jeśli jego wielkość będzie się powiększać.

System, na którym działa wirtualna maszyna Javy, musi udostępniać mechanizmy wyłącznego i współdzielonego blokowania dostępu do plików. Jeśli system operacyjny nie udostępnia możliwości tworzenia blokad współdzielonych, a program jej zażąda, zostanie utworzona blokada wyłączna. Rodzaj uzyskanej blokady (współdzielony lub wyłączny) można określić, korzystając z metody `FileLock.isShared()`.

Blokowanie dostępu do fragmentów pliku odwzorowanego w pamięci

Jak już wspominałem wcześniej, odwzorowywanie plików w pamięci jest zazwyczaj stosowane w przypadku obsługi bardzo dużych plików. Może się wówczas zdarzyć, że konieczne będzie zablokowanie dostępu do części takiego pliku, aby inne procesy mogły modyfikować jego niezablokowane fragmenty. Operacje tego typu są czasami wykonywane na bazach danych, aby ich zawartość mogła być jednocześnie dostępna dla wielu użytkowników.

Program przedstawiony w poniższym przykładzie uruchamia dwa wątki, z których każdy blokuje dostęp do innej części pliku:

```
//: io/LockingMappedFiles.java
// Blokowanie fragmentów pliku odwzorowanego w pamięci.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
```

```
public class LockingMappedFiles {
  static final int LENGTH = 0x8FFFFFF; // 128 MB
  static FileChannel fc;
  public static void main(String[] args) throws Exception {
    fc =
      new RandomAccessFile("test.dat", "rw").getChannel();
    MappedByteBuffer out =
      fc.map(FileChannel.MapMode.READ_WRITE, 0, LENGTH);
    for(int i = 0; i < LENGTH; i++)
      out.put((byte)'x');
    new LockAndModify(out, 0, 0 + LENGTH/3);
    new LockAndModify(out, LENGTH/2, LENGTH/2 + LENGTH/4);
  }
  private static class LockAndModify extends Thread {
    private ByteBuffer buff;
    private int start, end;
    LockAndModify(ByteBuffer mbb, int start, int end) {
      this.start = start;
      this.end = end;
      mbb.limit(end);
      mbb.position(start);
      buff = mbb.slice();
      start();
    }
    public void run() {
      try {
        // Blokada bez pokrywania:
        FileLock fl = fc.lock(start, end, false);
        System.out.println("Zablokowany od: "+ start +" do "+ end);
        // Przetwarzanie zablokowanej części:
        while(buff.position() < buff.limit() - 1)
          buff.put((byte)(buff.get() + 1));
        fl.release();
        System.out.println("Zwolniony od: "+start+" do "+ end);
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
  }
} ///:~
```

Klasa `LockAndModify`, dziedzicząca po klasie `Thread`, tworzy bufor fragmentu pliku, a następnie modyfikowany wycinek (posługując się metodą `slice()`). Następnie w metodzie `run()` blokowany jest dostęp do kanału skojarzonego z plikiem (nie można blokować dostępu do bufora — jedynie do kanału). Wywołanie metody `lock()` jest bardzo podobne do blokowania dostępu do obiektu w wątkach — tworzy ono „sekcję krytyczną" dysponującą wyłącznym dostępem do fragmentu pliku[5].

Blokady są automatycznie usuwane w momencie zamykania wirtualnej maszyny Javy lub kanału użytego do utworzenia blokady; można także jawnie wywołać metodę `release()` obiektu `FileLock`, jak pokazałem w powyższym przykładzie.

[5] Więcej o wątkach dowiesz się z rozdziału „Współbieżność".

Kompresja

Biblioteka wejścia-wyjścia Javy zawiera klasy obsługujące odczyt i zapis do strumieni w formacie skompresowanym. Można opakowywać w nie istniejące klasy wejścia-wyjścia, aby dostarczyć możliwość kompresji.

Klasy te nie wywodzą się z klasy `Reader` i `Writer`, ale są częścią hierarchii klasy `InputStream` i `OutputStream`. Powodem tego jest fakt, że biblioteka kompresji pracuje na bajtach, a nie znakach. Czasem jednak zachodzi potrzeba mieszania tych dwóch typów strumieni — pamiętajmy, że można użyć klasy `InputStreamReader` i `OutputStreamWriter`, aby zapewnić łatwą konwersję między jednym a drugim typem.

Klasa z biblioteki kompresji	Funkcja
`CheckedInputStream`	`getCheckSum()` produkuje sumę kontrolną dla dowolnego obiektu `InputStream` (nickoniecznie przy dekompresji)
`CheckedOutputStream`	`getCheckSum()` produkuje sumę kontrolną dla dowolnego obiektu `OutputStream` (niekoniecznie przy kompresji)
`DeflaterOutputStream`	Klasa bazowa dla klas kompresji
`ZipOutputStream`	Podklasa `DeflaterOutputStream` kompresująca dane do formatu pliku Zip
`GZIPOutputStream`	Podklasa `DeflaterOutputStream` kompresująca dane do formatu pliku GZIP
`InflaterInputStream`	Klasa bazowa dla klas dekompresji
`ZipInputStream`	Podklasa `InflaterInputStream` dekompresująca dane przechowywane w plikach Zip
`GZIPInputStream`	Podklasa `InflaterInputStream` dekompresująca dane przechowywane w plikach GZIP

Mimo że istnieje wiele algorytmów kompresji, prawdopodobnie właśnie Zip i GZIP są najczęściej stosowanymi. Łatwo więc można manipulować skompresowanymi w ten sposób danymi za pomocą wielu powszechnie dostępnych narzędzi.

Prosta kompresja do formatu GZIP

Interfejs GZIP jest prosty i dlatego prawdopodobnie najodpowiedniejszy do użycia w przypadku potrzeby skompresowania pojedynczego strumienia danych (w przeciwieństwie do pojemnika zawierającego zróżnicowane fragmenty danych). Poniższy przykład dokonuje kompresji pliku:

```
//: io/GZIPcompress.java
// {Args: GZIPcompress.java}
import java.util.zip.*;
import java.io.*;

public class GZIPcompress {
  public static void main(String[] args)
  throws IOException {
    if(args.length == 0) {
```

```
      System.out.println(
        "Stosowanie: \nGZIPcompress plik\n" +
        "\tKompresuje plik przy użyciu algorytmu GZIP: " +
        "plikiem wynikowym jest test.gz");
      System.exit(1);
    }
    BufferedInputStream in = new BufferedInputStream(
      new FileInputStream(args[0]));
    BufferedOutputStream out = new BufferedOutputStream(
      new GZIPOutputStream(
        new FileOutputStream("test.gz")));
    System.out.println("Zapis pliku");
    int c;
    while((c = in.read()) != -1)
      out.write(c);
    in.close();
    out.close();
    System.out.println("Odczyt pliku");
    BufferedReader in2 = new BufferedReader(
      new InputStreamReader(new GZIPInputStream(
        new FileInputStream("test.gz"))));
    String s;
    while((s = in2.readLine()) != null)
      System.out.println(s);
  }
} /* (Execute to see output) *///:~
```

Sposób użycia klas kompresji jest nieskomplikowany — po prostu opakowujemy strumień wyjściowy w obiekt `GZIPOutputStream` lub `ZipOutputStream`, natomiast strumień wejściowy w obiekt `GZIPInputStream` lub `ZipInputStream`. Wszystko poza tym odbywa się tak jak zwykły odczyt lub zapis wejścia-wyjścia.

Przechowywanie wielu plików w formacie Zip

Biblioteka obsługująca format Zip jest znacznie obszerniejsza. Za jej pomocą można łatwo przechowywać wiele plików naraz, istnieje nawet oddzielna klasa ułatwiająca odczytywanie archiwów Zip. Biblioteka wykorzystuje standardowy format Zip, współpracuje więc bez problemów ze wszystkimi narzędziami dostępnymi obecnie w internecie. Kolejny przykład podobny jest w formie do poprzedniego, ale obsługuje dowolną liczbę argumentów z wiersza poleceń. Oprócz tego demonstruje zastosowanie klas `Checksum` w celu obliczenia i zweryfikowania sumy kontrolnej pliku. Istnieją dwa typy `Checksum`: `Adler32` (szybsza) oraz `CRC32` (wolniejsza, ale odrobinę bardziej dokładna).

```
//: io/ZipCompress.java
// Użycie algorytmu ZIP do kompresji dowolnej liczby
// plików podanych w wierszu wywołania programu.
// {Args: ZipCompress.java}
import java.util.zip.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ZipCompress {
  public static void main(String[] args)
  throws IOException {
```

```
    FileOutputStream f = new FileOutputStream("test.zip");
    CheckedOutputStream csum =
      new CheckedOutputStream(f, new Adler32());
     ZipOutputStream zos = new ZipOutputStream(csum);
     BufferedOutputStream out =
      new BufferedOutputStream(zos);
    zos.setComment("Test kompresji w Javie");
    // Brak odpowiedniego wywołania getComment().
    for(String arg : args) {
      print("Zapis pliku " + arg);
      BufferedInputStream in =
        new BufferedInputStream(new FileInputStream(arg));
      zos.putNextEntry(new ZipEntry(arg));
      int c;
      while((c = in.read()) != -1)
        out.write(c);
      in.close();
      out.flush();
    }
    out.close();
    // Sumy kontrolne są ważne dopiero po zamknięciu pliku!
    print("Suma kontrolna: " + csum.getChecksum().getValue());
    // A teraz wydobycie plików:
    print("Odczyt pliku");
    FileInputStream fi = new FileInputStream("test.zip");
    CheckedInputStream csumi =
      new CheckedInputStream(fi, new Adler32());
    ZipInputStream in2 = new ZipInputStream(csumi);
    BufferedInputStream bis = new BufferedInputStream(in2);
    ZipEntry ze;
    while((ze = in2.getNextEntry()) != null) {
      print("Odczyt pliku " + ze);
      int x;
      while((x = bis.read()) != -1)
        System.out.write(x);
    }
    if(args.length == 1)
    print("Suma kontrolna: " + csumi.getChecksum().getValue());
    bis.close();
    // Alternatywny sposób otwierania i wczytywania archiwów Zip:
    ZipFile zf = new ZipFile("test.zip");
    Enumeration e = zf.entries();
    while(e.hasMoreElements()) {
      ZipEntry ze2 = (ZipEntry)e.nextElement();
      print("Plik: " + ze2);
      // ... tu wydobywanie danych jak poprzednio
    }
    /* if(args.length == 1) */
  }
} /* (Execute to see output) *///:~
```

Dla każdego pliku dodawanego do archiwum należy wywołać metodę `putNextEntry()` i przekazać jej obiekt `ZipEntry`. Obiekt `ZipEntry` zawiera obszerny interfejs, pozwalający na odczytanie i ustawienie danych dostępnych dla konkretnego wpisu z archiwum Zip: jego nazwy, rozmiaru przed i po kompresji, daty, sumy kontrolnej, dodatkowego pola danych, komentarza, sposobu kompresji i czy jest to katalog czy plik. Mimo że format

Zip umożliwia zabezpieczenie hasłem, opcja ta nie jest obsługiwana przez bibliotekę Javy. I chociaż `CheckedInputStream` i `CheckedOutputStream` obsługują oba typy sum kontrolnych (`Adler32` i `CRC32`), to `ZipEntry` zapewnia jedynie interfejs `CRC`. Jest to restrykcja formatu Zip, która ogranicza stosowanie szybszego sposobu `Adler32`.

Aby wydobywać pliki z archiwów, `ZipInputStream` dostarcza metodę `getNextEntry()`, która zwraca następny obiekt `ZipEntry`, jeśli taki istnieje. Bardziej zwięzłą w użyciu metodą jest odczytywanie pliku za pomocą obiektu `ZipFile`, który ma metodę `entries()` zwracającą obiekt typu `Enumeration`, zawierający obiekty `ZipEntry`.

W celu odczytania sumy kontrolnej musimy mieć dostęp do skojarzonego obiektu `Checksum`. Tutaj przechowujemy referencję do obiektów `CheckedOutputStream` i `CheckedInputStream`, jednak równie dobrze mogłaby to być referencja do obiektu `Checksum`.

Dezorientującą metodą strumieni Zip jest `setComment()`. Jak pokazałem wcześniej, możemy dodać komentarz na etapie kompresji pliku do archiwum, nie ma jednak możliwości odzyskania go za pomocą `ZipInputStream`. Wygląda na to, że komentarze są w pełni obsługiwane jedynie przy odczycie „wpis po wpisie", kiedy korzysta się z `ZipEntry`.

Oczywiście biblioteki Zip i GZIP nie ograniczają się tylko do plików — kompresować za ich pomocą możemy wszystko, łącznie z danymi przeznaczonymi do wysłania za pośrednictwem łącza internetowego[6].

Archiwa Javy (JAR)

Format Zip jest wykorzystywany także przez własny format Javy — JAR (Java ARchive), który służy do zebrania grupy plików do pojedynczego, skompresowanego pliku, tak jak w przypadku Zip. Jednak, jak wszystko w Javie, format JAR jest niezależny od platformy, nie musimy się więc martwić o reprezentację danych. Do plików JAR możemy włączać także pliki graficzne i audio, tak samo jak pliki klas.

Korzystanie z plików JAR jest szczególnie wygodne przy zastosowaniach internetowych. Przed wprowadzeniem tego formatu przeglądarka sieciowa musiała ponawiać żądania ściągnięcia z serwera wszystkich plików klas składających się na aplet. W dodatku wszystkie te pliki były nieskompresowane. Przy połączeniu wszystkich klas niezbędnych dla działania jednego apletu w jeden plik JAR konieczne jest tylko jedno żądanie wysłane do serwera, a transfer jest szybszy z powodu kompresji. W celu zachowania bezpieczeństwa każdy wpis w pliku JAR może być zaopatrzony w cyfrowy podpis.

Archiwum JAR składa się z pojedynczego pliku zawierającego zbiór plików skompresowanych oraz „wykaz" (ang. *manifest*), który je wszystkie opisuje (możemy utworzyć własny wykaz, ale jeśli tego nie zrobimy, wyręczy nas program *jar*). Więcej informacji na temat wykazów JAR można znaleźć w dokumentacji JDK.

Narzędzie *jar*, rozpowszechniane razem z JDK Suna, automatycznie kompresuje pliki wskazane przez użytkownika. Wywołuje się je z wiersza poleceń:

```
jar [opcje] plik_docelowy [wykaz] plik(i)
```

[6] Szczególnie strumienie `GZIP` (*in* i *out*) nadają się do przesyłania danych przez sieć. Wiele apletów korzysta z tej jakże prostej metody kompresowania danych pobieranych z dysku serwera WWW — *przyp. red.*

Opcje są po prostu zbiorem liter (nie są potrzebne myślniki ani żaden inny znacznik). Użytkownicy Uniksa i Linuksa odnajdą w nich podobieństwa do opcji programu *tar*. Oto one:

Opcja	Znaczenie
`c`	Tworzy nowe lub puste archiwum.
`t`	Wypisuje zawartość archiwum.
`x`	Wydobywa wszystkie pliki.
`x plik`	Wydobywa wskazany plik.
`f`	Wymusza podanie nazwy pliku. Jeśli brak tej opcji, *jar* przyjmuje, że dane będą pochodziły ze standardowego wejścia, lub, jeśli tworzy plik, że ma wysyłać dane na standardowe wyjście.
`m`	Oznacza, że pierwszy argument będzie nazwą wykazu stworzonego przez użytkownika.
`v`	Zwiększa liczbę prezentowanych informacji — *jar* będzie wypisywał na standardowe wyjście opis tego, co robi.
`0`	Jedynie przechowuje pliki, bez kompresji. Można jej użyć, aby stworzyć plik *JAR* i dołączyć go do zmiennej środowiskowej `CLASSPATH` (będzie on wtedy szybciej wczytywany).
`M`	Zapobiega automatycznemu tworzeniu wykazu.

Jeżeli wśród plików wskazanych do kompresji programowi *jar* znajduje się podkatalog, wszystkie zawarte w nim pliki i podkatalogi zostaną również automatycznie dołączone do archiwum. Zapamiętywana jest także informacja o ścieżkach.

Oto kilka typowych wywołań programu `jar`. Pierwsze polecenie tworzy archiwum *mojPlikJar.jar*, zawierające wszystkie pliki klas z bieżącego katalogu, razem z automatycznie wygenerowanym wykazem:

```
jar cf mojPlikJar.jar *.class
```

Podobnie jak w poprzednim przykładzie, ale z włączeniem wykazu stworzonego przez użytkownika:

```
jar cmf mojPlikJar.jar mojWykaz.mf *.class
```

Wypisze listę plików zawartych w *mojPlikJar.jar*:

```
jar tf mojPlikJar.jar
```

Dodano opcję `v`, aby uzyskać bardziej szczegółowe informacje o plikach:

```
jar tvf mojPlikJar.jar
```

Przy założeniu, że *audio*, *klasy* i *obrazy* to nazwy podkatalogów, pliki w nich zawarte zostaną włączone do pliku *mojAplet.jar*. Opcja `v` pozwala na obserwację działania programu:

```
jar cvf mojAplet.jar audio klasy obrazy
```

Archiwa JAR mogą zostać umieszczone w zmiennej środowiskowej `CLASSPATH`:

```
CLASSPATH="lib1.jar;lib2.jar"
```

Wtedy Java może szukać klas również w archiwach *lib1.jar* oraz *lib2.jar*.

Program *jar* nie jest tak użyteczny, jak narzędzia zip. Na przykład nie można dodać ani usunąć plików z istniejącego pliku *JAR* — można go jedynie stworzyć od zera. Nie można również przesuwać plików do archiwum *JAR*, usuwając je w miarę przenoszenia. Tym niemniej plik *JAR* stworzony na jednej platformie będzie prawidłowo odczytywany przez narzędzia *jar* na każdej innej (ten problem czasami dotyczy plików w formacie zip).

Jak zobaczymy w rozdziale „Graficzne interfejsy użytkownika", w plikach JAR przechowywane są komponenty JavaBeans[7].

Serializacja obiektów

Kiedy tworzysz w Javie obiekt, istnieje on dopóty, dopóki jest potrzebny w programie, ale bezwarunkowo przestanie istnieć po zakończeniu programu. Niby nic w tym niestosownego, są jednak sytuacje, kiedy chcielibyśmy, aby obiekt mógł istnieć — i przechowywać informacje — nawet gdy program nie działa; tak, aby przy następnym uruchomieniu programu obiekt był gotowy do użycia z kompletem danych, które miał przed zakończeniem programu. Oczywiście efekt taki można uzyskać, zapisując w odpowiednim momencie dane obiektu do pliku czy bazy danych, ale skoro już wszystko ma być obiektem, przydałaby się możliwość tworzenia też obiektów „trwałych" — aby dało się określić obiekt jako trwały, a jego utrwalenie złożyć na kompilator i środowisko wykonawcze.

Mechanizm *serializacji obiektów* w Javie pozwala zamienić dowolny obiekt, który implementuje interfejs `Serializable`, na sekwencję bajtów w sposób umożliwiający późniejsze wierne odtworzenie oryginalnego obiektu. Działa to nawet w sieci, co oznacza, że mechanizm serializacji automatycznie kompensuje różnice między systemami operacyjnymi. Pozwala to na stworzenie obiektu na komputerze działającym pod Windows, zserializowanie go i przesłanie siecią do maszyny uniksowej, gdzie zostanie prawidłowo odtworzony. Nie trzeba się już martwić o reprezentację danych w różnych systemach, uporządkowanie bajtów ani inne szczegóły.

Serializacja obiektów jest interesująca sama w sobie, ponieważ umożliwia zaimplementowanie *lekkiej trwałości* (ang. *lightweight persistence*). Trwałość oznacza, że czas życia obiektu nie jest ograniczony czasem działania programu — obiekt żyje pomiędzy kolejnymi jego uruchomieniami. Przez zapisanie zserializowanego obiektu na dysku i odtworzenie go przy kolejnym wywołaniu programu można osiągnąć efekt trwałości. Powód, dla którego określamy ją jako „lekką", to fakt, że nie da się po prostu zdefiniować obiektu, używając czegoś w rodzaju słowa kluczowego „persistent" i pozwolić systemowi zadbać o szczegóły (chociaż może będzie to możliwe w przyszłości). Zamiast tego musimy jawnie serializować i deserializować obiekty w programach. Jeśli potrzebujesz lepszego mechanizmu zapewniającego trwałość obiektów, warto przyjrzeć się narzędziom takim, jak *Hibernate* (*http://www.hibernate.org*). Szczegółowe informacje na ten temat można znaleźć w książce *Thinking in Enterprise Java*, do pobrania z witryny *www.MindView.net*.

[7] Gra słów: w języku angielskim słowo *jar* oznacza również *słój*, *beans* to ziarna, a *Java* jest też odmianą kawy — *przyp. tłum.*

Serializacja obiektów została dodana do języka w celu obsługi dwóch ważnych możliwości. *Zdalne wywoływanie metod* (RMI — *Remote Method Invocation*) Javy pozwala obiektowi istniejącemu na jednym komputerze zachowywać się tak, jakby w rzeczywistości istniał na drugim. Przy wysyłaniu komunikatów do odległego obiektu, serializacja danych jest konieczna, aby przekazywać argumenty metod i wartości zwracane. Omówienie RMI można znaleźć w książce *Thinking in Enterprise Java*.

Serializacja jest również niezbędna dla komponentów JavaBeans, opisanych w rozdziale „Graficzne interfejsy użytkownika". Kiedy Bean (dosłownie *ziarno*, zwykle jednak używane jest pojęcie *komponent*) jest używany, jego stan jest konfigurowany w czasie projektowania. Informacja o jego stanie musi być zapamiętana i odtworzona potem, kiedy zostaje uruchomiony program; możliwość tę zapewnia właśnie serializacja obiektów.

Serializacja obiektu jest całkiem prosta, przy założeniu, że obiekt implementuje interfejs `Serializable` (jest to wyłącznie interfejs znakujący i nie posiada żadnych metod). Kiedy została dodana do języka, wiele standardowych klas bibliotecznych zmodyfikowano tak, aby nadawały się do serializacji, włączając w to wszystkie klasy opakowujące dla typów podstawowych, wszystkie klasy kontenerowe i wiele innych. Nawet obiekt `Class` może być serializowany.

Aby zserializować obiekt, należy utworzyć jakiś rodzaj obiektu `OutputStream`, a następnie opakować go w obiekt `ObjectOutputStream` (serializacja obiektów jest zorientowana *bajtowo*, korzysta więc z hierarchii klasy `InputStream` i `OutputStream`). W tym momencie wystarczy jedynie wywołać metodę `writeObject()`, żeby zserializowany obiekt został wysłany do `OutputStream`. W celu odwrócenia tego procesu opakowujemy `InputStream` w obiekt `ObjectInputStream` i korzystamy z metody `readObject()`. Otrzymuje się w ten sposób referencję do obiektu klasy bazowej `Object`, zatem trzeba dokonać odpowiedniego rzutowania, aby uzyskać pożądany efekt.

Szczególnie ciekawą właściwością serializacji jest to, że zapisywany jest nie tylko obraz pierwotnego obiektu. Jeżeli zawiera on referencje do innych obiektów, to zostaną zapisane również *te* obiekty — a jeśli one zawierają referencje, to kolejne obiekty itd. Powstałą strukturę określa się czasem mianem „sieci obiektów", z którą może być połączony pojedynczy obiekt, i zawiera ona zarówno tablice referencji, jak i wszystkie części składowe. Gdyby użytkownik musiał tworzyć własne schematy serializacji, utworzenie kodu obsługującego wszystkie te połączenia byłoby dość skomplikowane. Wydaje się jednak, że serializacja obiektów Javy radzi sobie z tym doskonale, używając zoptymalizowanego algorytmu przebiegającego całą sieć obiektów. Następujący przykład testuje mechanizm serializacji, konstruując „dżdżownicę" (klasa `Worm`) połączonych obiektów, przy czym każdy z nich zawiera wskaźnik do następnego segmentu oraz tablicę referencji do obiektów klasy `Data`:

```
//: io/Worm.java
// Demonstracja serializacji obiektów.
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Data implements Serializable {
  private int n;
  public Data(int n) { this.n = n; }
  public String toString() { return Integer.toString(n); }
```

```
}

public class Worm implements Serializable {
  private static Random rand = new Random(47);
  private Data[] d = {
    new Data(rand.nextInt(10)),
    new Data(rand.nextInt(10)),
    new Data(rand.nextInt(10))
  };
  private Worm next;
  private char c;
  // Wartość i to liczba segmentów
  public Worm(int i, char x) {
    print("Konstruktor Worm: " + i);
    c = x;
    if(--i > 0)
      next = new Worm(i, (char)(x + 1));
  }
  public Worm() {
    print("Konstruktor domyślny");
  }
  public String toString() {
    StringBuilder result = new StringBuilder(":");
    result.append(c);
    result.append("(");
    for(Data dat : d)
      result.append(dat);
    result.append(")");
    if(next != null)
      result.append(next);
    return result.toString();
  }
  public static void main(String[] args)
  throws ClassNotFoundException, IOException {
    Worm w = new Worm(6, 'a');
    print("w = " + w);
    ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream("worm.out"));
    out.writeObject("Zapis Worm\n");
    out.writeObject(w);
    out.close(); // Z opróżnieniem bufora wyjściowego
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("worm.out"));
    String s = (String)in.readObject();
    Worm w2 = (Worm)in.readObject();
    print(s + "w2 = " + w2);
    ByteArrayOutputStream bout =
      new ByteArrayOutputStream();
    ObjectOutputStream out2 = new ObjectOutputStream(bout);
    out2.writeObject("Zapis Worm\n");
    out2.writeObject(w);
    out2.flush();
    ObjectInputStream in2 = new ObjectInputStream(
      new ByteArrayInputStream(bout.toByteArray()));
    s = (String)in2.readObject();
    Worm w3 = (Worm)in2.readObject();
    print(s + "w3 = " + w3);
  }
```

```
} /* Output:
Konstruktor Worm: 6
Konstruktor Worm: 5
Konstruktor Worm: 4
Konstruktor Worm: 3
Konstruktor Worm: 2
Konstruktor Worm: 1
w = :a(853):b(119):c(802):d(788):e(199):f(881)
Zapis Worm
w2 = :a(853):b(119):c(802):d(788):e(199):f(881)
Zapis Worm
w3 = :a(853):b(119):c(802):d(788):e(199):f(881)
*///:~
```

Aby było ciekawiej, tablica obiektów `Data` wewnątrz dżdżownicy jest inicjalizowana przypadkowymi liczbami (w ten sposób nie można podejrzewać kompilator o przechowywanie jakiejś metainformacji). Każdy segment `Worm` jest etykietowany wartością typu `char`, która zostaje wygenerowana automatycznie w procesie rekurencyjnego tworzenia połączonej listy obiektów `Worm`. Argumentem dla konstruktora jest to, jak długa ma być nasza „dżdżownica" — aby uzyskać referencję `next`, konstruktor zostaje wywołany rekurencyjnie z długością o jeden mniejszą. Ostatnia referencja `next` jest równa `null` i oznacza koniec robaczka.

Celem takiej konstrukcji było zaprojektowanie w miarę złożonego obiektu, który nie nadawałby się do serializacji w łatwy sposób. Jednak uruchomić serializację można bardzo prosto. Kiedy już utworzymy `ObjectOutputStream` z jakiegoś innego strumienia, metoda `writeObject()` serializuje obiekt. Zwróć także uwagę na wywołanie `writeObject()` obiektu `String`. Można zapisywać również wszystkie podstawowe typy danych, korzystając z tych samych metod, co w klasie `DataOutputStream` (dzielą one ten sam interfejs).

W kodzie widać dwa oddzielne fragmenty, które wyglądają podobnie. Pierwszy zapisuje i odczytuje plik, natomiast drugi, dla odmiany, zapisuje i odczytuje obiekt `ByteArray`. Przy użyciu serializacji można wysłać i odebrać plik z dowolnego strumienia `DataInputStream` lub `DataOutputStream`, włączając w to połączenie sieciowe, o czym można się przekonać w książce *Thinking in Enterprise Java*.

Widać też, że obiekt odzyskany po serializacji rzeczywiście zawiera wszystkie połączenia oryginalnego obiektu.

Zauważmy, że żaden konstruktor (nawet domyślny) nie jest wywoływany w procesie deserializacji obiektu `Serializable`. Cały obiekt jest odtwarzany przez odczytanie informacji ze strumienia `InputStream`.

Ćwiczenie 27. Utwórz klasę implementującą interfejs `Serializable` zawierającą referencję do obiektu innej implementacji `Serializable`. Utwórz egzemplarz własnej klasy, utrwal go na dysku, a następnie przywróć i sprawdź, czy całość przebiegła poprawnie (1).

Odnajdywanie klasy

Zastanówmy się, jakie informacje są niezbędne, aby odzyskać zserializowany obiekt. Na przykład przypuśćmy, że serializujemy obiekt i wysyłamy go jako plik albo za pośrednictwem sieci do innego komputera. Czy program działający na tym komputerze może zrekonstruować obiekt, polegając jedynie na zawartości tego pliku?

Najlepszym sposobem odpowiedzi na to pytanie jest (jak zwykle) przeprowadzenie eksperymentu. W podkatalogu kodu źródłowego dla tego rozdziału znajduje się następujący plik:

```
//: io/Alien.java
// Klasa serializowana.
import java.io.*;
public class Alien implements Serializable {} ///:~
```

W tym samym katalogu znajduje się plik, który tworzy i serializuje obiekt osobnika obcego `Alien`:

```
//: io/FreezeAlien.java
// Zapis serializowanego obiektu do pliku.
import java.io.*;

public class FreezeAlien {
  public static void main(String[] args) throws Exception {
    ObjectOutput out = new ObjectOutputStream(
      new FileOutputStream("X.file"));
    Alien quellek = new Alien();
    out.writeObject(quellek);
  }
} ///:~
```

Zamiast przechwytywać i obsługiwać wyjątki, ten program stosuje szybszy i nieelegancki sposób — pozwala im wypaść poza `main()` tak, że informacje o nich będą wypisane na konsolę.

Po skopiowaniu i uruchomieniu program tworzy w katalogu *io* plik o nazwie *X.file*. Poniższy kod znajduje się natomiast w podkatalogu *xfiles*:

```
//: io/xfiles/ThawAlien.java
// Próba odtworzenia obiektu serializowanego w pliku
// bez znajomości klasy tego obiektu.
// {RunByHand}
import java.io.*;

public class ThawAlien {
  public static void main(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream(new File("..", "X.file")));
    Object mystery = in.readObject();
    System.out.println(mystery.getClass());
  }
} /* Output:
class Alien
*///:~
```

Nawet otworzenie pliku i wczytanie obiektu `mystery` wymaga obiektu `Class` dla obiektu `Alien`; jednak wirtualna maszyna Javy (JVM) nie może znaleźć pliku *Alien.class* (chyba że znajdzie się on w jednym z katalogów wymienionych w zmiennej środowiskowej `CLASSPATH`, co w naszym przypadku nie powinno się zdarzyć). Otrzymamy wyjątek `ClassNotFoundException` (i znów cały materiał dowodowy dotyczący istnienia obcych znika, zanim mógłby zostać zweryfikowany!). Wirtualna maszyna Javy musi mieć dostęp do niezbędnych plików *.class*.

Kontrola serializacji

Jak widać, domyślny mechanizm serializacji jest prosty w użyciu. Co jednak zrobić w przypadku dodatkowych wymagań? Być może dla bezpieczeństwa nie chcemy serializować pewnych części obiektu bądź też nie ma sensu serializować podobiektów, jeżeli i tak będą musiały być stworzone od nowa po odtworzeniu obiektu.

Procesem serializacji można sterować, implementując interfejs `Externalizable` zamiast `Serializable`. `Externalizable` jest rozszerzeniem interfejsu `Serializable` o dwie metody, `writeExternal()` i `readExternal()`, które są wywoływane automatycznie przy serializacji i deserializacji obiektu, tak aby można było wykonać specjalne operacje.

Następujący przykład pokazuje prostą implementację metod interfejsu `Externalizable`. Zauważmy, że `Blip1` i `Blip2` są niemal identyczne poza jedną małą różnicą (sprawdź, czy możesz ją odkryć, patrząc na kod):

```
//: io/Blips.java
// Proste zastosowanie interfejsu Externalizable i mała pułapka.
import java.io.*;
import static net.mindview.util.Print.*;

class Blip1 implements Externalizable {
  public Blip1() {
    print("Konstruktor Blip1");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    print("Blip1.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    print("Blip1.readExternal");
  }
}

class Blip2 implements Externalizable {
  Blip2() {
    print("Konstruktor Blip2");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    print("Blip2.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    print("Blip2.readExternal");
  }
}

public class Blips {
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    print("Konstrukcja obiektów:");
    Blip1 b1 = new Blip1();
    Blip2 b2 = new Blip2();
```

```
    ObjectOutputStream o = new ObjectOutputStream(
      new FileOutputStream("Blips.out"));
    print("Zapis obiektów:");
    o.writeObject(b1);
    o.writeObject(b2);
    o.close();
    // A teraz odtworzenie:
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("Blips.out"));
    print("Odtwarzanie b1:");
    b1 = (Blip1)in.readObject();
    // OHO! Wyjątek:
//! print("Odtwarzanie b2:");
//! b2 = (Blip2)in.readObject();
  }
} /* Output:
Konstrukcja obiektów:
Konstruktor Blip1
Konstruktor Blip2
Zapis obiektów:
Blip1.writeExternal
Blip2.writeExternal
Odtwarzanie b1:
Konstruktor Blip1
Blip1.readExternal
*///:~
```

Obiekt `Blip2` nie został odzyskany, ponieważ próba jego odtworzenia spowodowała wyjątek. Czy zauważyłeś różnicę między `Blip1` a `Blip2`? Konstruktor `Blip1` jest publiczny, podczas gdy konstruktor `Blip2` nie jest i to właśnie powoduje zgłoszenie wyjątku przy próbie odzyskania `Blip2`. Spróbuj upublicznić konstruktor `Blip2` i usunąć komentarze `//!`, aby obejrzeć prawidłowe działanie programu.

Przy rekonstrukcji `b1` wywoływany jest domyślny konstruktor `Blip1`. Jest to różnica w porównaniu z obiektem `Serializable`, który jest odtwarzany wyłącznie na podstawie jego zapisu i nie ma miejsca wywołanie konstruktora. W przypadku obiektu `Externalizable` zachodzą zwykłe działania dla etapu konstrukcji (łącznie z inicjalizacją zmiennych w momencie ich definiowania), a *potem* zostaje wywołana metoda `readExternal()`. Trzeba mieć tego świadomość — szczególnie tego, że zawsze ma miejsce domyślna konstrukcja obiektu — aby osiągnąć pożądane zachowanie obiektu `Externalizable`.

Poniższy przykład pokazuje, co trzeba zrobić, aby w pełny sposób przechować i odtworzyć obiekt implementujący `Externalizable`.

```
//: io/Blip3.java
// Rekonstrukcja obiektów Externalizable.
import java.io.*;
import static net.mindview.util.Print.*;

public class Blip3 implements Externalizable {
  private int i;
  private String s; // Bez inicjalizacji
  public Blip3() {
    print("Konstruktor Blip3");
    // s oraz i niezainicjalizowane
  }
```

```
  public Blip3(String x, int a) {
    print("Blip3(String x, int a)");
    s = x;
    i = a;
    // s oraz i inicjalizowane jedynie w konstruktorze
    // innym niż domyślny.
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput out)
  throws IOException {
    print("Blip3.writeExternal");
    // Koniecznie:
    out.writeObject(s);
    out.writeInt(i);
  }
  public void readExternal(ObjectInput in)
  throws IOException, ClassNotFoundException {
    print("Blip3.readExternal");
    // Koniecznie:
    s = (String)in.readObject();
    i = in.readInt();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    print("Konstrukcja obiektów:");
    Blip3 b3 = new Blip3("Ciąg znaków ", 47);
    print(b3);
    ObjectOutputStream o = new ObjectOutputStream(
      new FileOutputStream("Blip3.out"));
    print("Zapis obiektów:");
    o.writeObject(b3);
    o.close();
    // A teraz odtworzenie:
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("Blip3.out"));
    print("Odtwarzanie b3:");
    b3 = (Blip3)in.readObject();
    print(b3);
  }
} /* Output:
Konstrukcja obiektów:
Blip3(String x, int a)
Ciąg znaków 47
Zapis obiektów:
Blip3.writeExternal
Odtwarzanie b3:
Konstruktor Blip3
Blip3.readExternal
Ciąg znaków 47
*///:~
```

Pola `s` oraz `i` są inicjalizowane jedynie w drugim konstruktorze, nie w konstruktorze domyślnym. To oznacza, że jeśli nie zainicjalizujemy ich w metodzie `readExternal()`, będą miały wartość `null` (ponieważ obszar pamięci obiektu zostaje wymazany w pierwszym kroku tworzenia obiektu). Po oznaczeniu jako komentarz dwóch wierszy kodu następujących po słowach „Koniecznie:" i uruchomieniu programu okaże się, że po odzyskaniu obiektu `s` ma wartość `null`, a `i` zero.

Jeśli dziedziczymy po obiekcie `Externalizable`, przeważnie będziemy wywoływać wersje metody `writeExternal()` i `readExternal()` klasy bazowej, aby zapewnić prawidłowe zapamiętanie i odtworzenie składowych klasy bazowej.

Aby zatem wszystko działało poprawnie, należy nie tylko zapisać ważne dane obiektu za pomocą metody `writeExternal()` (nie ma żadnego domyślnego mechanizmu zapisującego jakąkolwiek składową obiektu `Externalizable`), ale także samodzielnie odzyskać dane metodą `readExternal()`. Z początku może to być nieco mylące, ponieważ zjawisko domyślnej konstrukcji obiektu `Externalizable` sugeruje, że jakiś rodzaj zapisu i odtwarzania zachodzi automatycznie. Tak nie jest.

Ćwiczenie 28. Skopiuj plik *Blips.java* i przemianuj go na *BlipCheck.java*. Zmień nazwę klasy `Blip2` na `BlipCheck` (uczyń ją publiczną i usuń z zakresu publicznego klasę `Blips`). Usuń z pliku oznaczenia `//!` i wykonaj program. Następnie wykomentuj konstruktor domyślny dla `BlipCheck`. Uruchom program i wyjaśnij, dlaczego działa. Zauważ, że po kompilacji musisz wykonać program przez `java Blips`, ponieważ metoda `main()` jest wciąż w klasie `Blips` (2).

Ćwiczenie 29. W pliku *Blip3.java* wykomentuj dwa wiersze po słowach „Koniecznie:" i uruchom program. Wyjaśnij otrzymane wyniki i to, dlaczego różnią się one od wyników wersji bez komentarza (2).

Słowo kluczowe transient

Kiedy kontrolujemy serializację, może istnieć jakiś szczególny podobiekt, którego nie chcemy zapisywać i odtwarzać automatycznie za pomocą mechanizmu serializacji Javy. Najczęściej jest tak w przypadku, kiedy podobiekt reprezentuje jakąś wrażliwą informację, której nie chcemy w ogóle serializować, taką jak np. hasło. Nawet kiedy w naszym obiekcie ta informacja jest prywatna, to jeżeli zostanie on poddany serializacji, ktoś może ją odtworzyć, odczytując plik lub przechwytując transmisję sieciową.

Jednym ze sposobów zapobiegania zapisaniu wrażliwych części obiektu jest zaimplementowanie swojej klasy jako `Externalizable`, tak jak widzieliśmy wcześniej. Wtedy nic nie jest automatycznie serializowane i można w sposób jawny zapisywać tylko niezbędne części obiektu wewnątrz metody `writeExternal()`.

Kiedy jednak pracujemy z obiektem `Serializable`, cały proces serializacji przebiega automatycznie. Aby go kontrolować, możemy wyłączyć serializację dla konkretnych pól, stosując słowo kluczowe `transient` — mówi ono: „Nie zawracaj sobie głowy zapisywaniem i odczytywaniem tego, zajmę się tym sam".

Dla przykładu rozważmy obiekt `Logon`, który przechowuje dane o sesji internetowej. Przypuśćmy, że po zweryfikowaniu praw dostępu chcemy przechować dane użytkownika, ale bez hasła. Najprościej można to osiągnąć, implementując interfejs `Serializable` i poprzedzając pole `password` słowem kluczowym `transient`. Kod wygląda następująco:

```
//: io/Logon.java
// Demonstracja użycia słowa "transient".
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;
```

```
public class Logon implements Serializable {
  private Date date = new Date();
  private String username;
  private transient String password;
  public Logon(String name, String pwd) {
    username = name;
    password = pwd;
  }
  public String toString() {
    return "logowanie: \n   użytkownik: " + username +
      "\n   data: " + date + "\n   hasło: " + password;
  }
  public static void main(String[] args) throws Exception {
    Logon a = new Logon("Kolos", "TomcioPaluch");
    print("logowanie a = " + a);
    ObjectOutputStream o = new ObjectOutputStream(
      new FileOutputStream("Logon.out"));
    o.writeObject(a);
    o.close();
    TimeUnit.SECONDS.sleep(1); // Opóźnienie
    // A teraz odtwarzanie:
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("Logon.out"));
    print("Odtwarzanie obiektu, czas: " + new Date());
    a = (Logon)in.readObject();
    print("logowanie a = " + a);
  }
} /* Output: (Sample)
logowanie a = logowanie:
  użytkownik: Kolos
  data: Fri Mar 17 11:59:13 CET 2006
  hasło: TomcioPaluch
Odtwarzanie obiektu, czas: Fri Mar 17 11:59:14 CET 2006
logowanie a = logowanie:
  użytkownik: Kolos
  data: Fri Mar 17 11:59:13 CET 2006
  hasło: null
*///:~
```

Jak widać, `date` i `username` są zwyczajnymi polami (nie `transient`), zatem podlegają automatycznej serializacji. Pole `password` jest jednak poprzedzone słowem kluczowym `transient` i dlatego nie jest zapisywane na dysku; mechanizm serializacji nie próbuje go także odzyskać. Kiedy obiekt zostaje odzyskany, wartość pola `password` to `null`. Zauważmy, że kiedy metoda `toString()` składa ciąg znaków `String` za pomocą przeciążonego dla `String` operatora '+', referencja pusta jest automatycznie konwertowana na ciąg „`null`".

Widać również, że pole `date` jest zapisywane i odtwarzane z dysku, a nie tworzone od nowa.

Ponieważ obiekty `Externalizable` domyślnie nie zapisują żadnych pól, słowa kluczowego `transient` używa się jedynie z obiektami `Serializable`.

Alternatywa dla Externalizable

Jeżeli nie podoba się komuś perspektywa implementowania interfejsu `Externalizable`, istnieje jeszcze inne rozwiązanie. Można zaimplementować interfejs `Serializable` i *dodać* (zauważ, że napisałem „dodać", a nie „przesłonić" czy „zaimplementować") metody

nazwane `writeObject()` i `readObject()`, które będą wywoływane automatycznie, kiedy obiekt będzie odpowiednio serializowany lub deserializowany. Oznacza to, że w przypadku dostarczenia tych dwóch metod zostaną one użyte zamiast domyślnej serializacji.

Metody te muszą zostać zadeklarowane dokładnie tak:

```
private void writeObject(ObjectOutputStream stream)
throws IOException;

private void readObject(ObjectInputStream stream)
throws IOException, ClassNotFoundException;
```

Z punktu widzenia projektowania dzieją się tu naprawdę dziwne rzeczy. Po pierwsze, można by pomyśleć, że skoro metody te nie są częścią klasy bazowej ani interfejsu `Serializable`, to powinny zostać zdefiniowane w swoich własnych interfejsach (interfejsie). Zauważmy także, że są one zdefiniowane jako prywatne (`private`), co oznacza, że dostęp do nich mają tylko obiekty tej klasy. Naprawdę jednak nie będą one wcale wywoływane przez obiekty tej klasy, ale przez obiekty klasy `ObjectOutputStream` i `ObjectInputStream` zamiast ich własnych metod `writeObject()` i `readObject()` (z olbrzymim wysiłkiem powstrzymuję się tu od długiego wywodu na temat użycia tych samych nazw. Krótko mówiąc, to bardzo mylące). Można się też zastanawiać, w jaki sposób `ObjectOutputStream` i `ObjectInputStream` mają dostęp do prywatnych metod naszej klasy. Możemy jedynie założyć, że jest to część magii procesu serializacji[8].

Każda metoda zdefiniowana w interfejsie jest automatycznie publiczna, skoro więc `readObject()` i `writeObject()` muszą być prywatne, nie mogą być częścią interfejsu. Ale ponieważ jesteśmy zmuszeni do zastosowania podanych sygnatur, efekt jest taki, jakbyśmy jednak zaimplementowali interfejs.

Wygląda na to, że kiedy wywoływana jest `ObjectOutputStream.writeObject()`, obiekt `Serializable`, który do niej przekazujemy, jest pytany (bez wątpienia przy zastosowaniu mechanizmu refleksji), czy implementuje własną metodę `writeObject()`. Jeśli tak, omijany jest zwykły proces serializacji i następuje wywołanie `writeObject()`. Analogicznie dzieje się w przypadku wywołania `readObject()`.

Jest jeszcze jeden trik. Wewnątrz własnej metody `writeObject()` możemy zdecydować się na użycie domyślnej `writeObject()`, stosując wywołanie `defaultWriteObject()`. Podobnie z wnętrza `readObject()` można wywołać `defaultReadObject()`. Następujący prosty przykład pokazuje, jak można kontrolować sposób przechowywania i odzyskiwania obiektu `Serializable`:

```
//: io/SerialCtl.java
// Kontrolowanie serializacji za pomocą własnych
// metod writeObject() i readObject().
import java.io.*;

public class SerialCtl implements Serializable {
  private String a;
  private transient String b;
  public SerialCtl(String aa, String bb) {
```

[8] Wyjaśnienie tej iście prestidigitatorskiej sztuczki zawiera rozdział „Informacje o typach" w podrozdziale „Interfejsy a RTTI".

```
    a = "Pole zwykłe: " + aa;
    b = "Pole transient: " + bb;
  }
  public String toString() { return a + "\n" + b; }
  private void writeObject(ObjectOutputStream stream)
  throws IOException {
    stream.defaultWriteObject();
    stream.writeObject(b);
  }
  private void readObject(ObjectInputStream stream)
  throws IOException, ClassNotFoundException {
    stream.defaultReadObject();
    b = (String)stream.readObject();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    SerialCtl sc = new SerialCtl("Test1", "Test2");
    System.out.println("Przed:\n" + sc);
    ByteArrayOutputStream buf= new ByteArrayOutputStream();
    ObjectOutputStream o = new ObjectOutputStream(buf);
    o.writeObject(sc);
    // A teraz odtwarzanie:
    ObjectInputStream in = new ObjectInputStream(
      new ByteArrayInputStream(buf.toByteArray()));
    SerialCtl sc2 = (SerialCtl)in.readObject();
    System.out.println("Po:\n" + sc2);
  }
} /* Output:
Przed:
Pole zwykłe: Test1
Pole transient: Test2
Po:
Pole zwykłe: Test1
Pole transient: Test2
*///:~
```

W przykładzie tym jedno pole `String` jest zwyczajne, a drugie `transient`, w celu udowodnienia, że pola nieoznaczone jako `transient` zapisywane są przez metodę `defaultWriteObject()`, natomiast pole `transient` jest zapisywane i odzyskiwane jawnie. Pola są inicjalizowane wewnątrz konstruktora, a nie w miejscu definicji, aby pokazać, że żaden automatyczny mechanizm nie inicjalizuje ich w czasie deserializacji.

Jeżeli mamy zamiar użyć domyślnego mechanizmu, aby zapisać części obiektu, które nie są `transient`, musimy jako pierwszą operację wewnątrz `writeObject()` wywołać `defaultWriteObject()` i jako pierwszą operację w `readObject()` wywołać `defaultReadObject()`. Jest to dziwne działanie — wydawałoby się na przykład, że wywołujemy `defaultWriteObject()` na rzecz obiektu `ObjectOutputStream` i nie przekazujemy jej żadnych argumentów, a jednak w jakiś sposób zna ona referencję do naszego obiektu i wie, jak go zapisać. Trochę niesamowite.

Przechowywanie i odtwarzanie obiektów `transient` wymaga kodu wyglądającego bardziej znajomo. Zastanówmy się jednak, co naprawdę się tutaj dzieje. W `main()` obiekt `SerialCtl` jest tworzony, a potem serializowany do strumienia `ObjectOutputStream` (w tym przypadku użyto bufora, a nie pliku — obiektowi `ObjectOutputStream` jest wszystko jedno). Serializacja następuje w wierszu:

```
o.writeObject(sc);
```

Metoda `writeObject()` musi zbadać sc, aby dowiedzieć się, czy ma on własną metodę `writeObject()` (nie przez sprawdzenie interfejsu — bo go nie ma — lub typu klasy, ale poszukując za pomocą mechanizmu refleksji). Jeśli ją znajdzie, to z niej korzysta. Podobnie dzieje się w przypadku metody `readObject()`. Być może był to jedyny praktyczny sposób rozwiązania problemu, ale z pewnością jest on dziwny.

Różnicowanie wersji klas

Możliwe, że ktoś będzie chciał utworzyć nową wersję klasy serializowalnej (na przykład dlatego, że obiekty oryginalnej klasy są przechowywane w bazie danych). Chociaż Java stwarza taką możliwość, wykorzystuje się ją jednak w nielicznych przypadkach i wymaga to głębszej znajomości języka niż zapewniana przez tę książkę. Dokumentacja JDK dostępna w witrynie *http://java.sun.com* zawiera obszerne omówienie tego tematu.

W dokumentacji JDK można zauważyć wiele komentarzy rozpoczynających się w sposób:

> ***Warning:*** *Serialized objects of this class will not be compatible with future Swing releases. The current serialization support is appropriate for short term storage or RMI between applications.*
>
> *(**Ostrzeżenie:** Serializowane obiekty tej klasy nie będą zgodne z przyszłymi wersjami Swing. Obecna obsługa serializacji jest odpowiednia do przechowywania krótkoterminowego lub wywołań RMI pomiędzy aplikacjami).*

Dzieje się tak, ponieważ mechanizm obsługi wersji jest zbyt prosty, aby pracować poprawnie we wszystkich sytuacjach, szczególnie z komponentami JavaBeans. Trwają prace nad ulepszeniem projektu i właśnie tego dotyczy ostrzeżenie.

Stosowanie trwałości

Użycie technologii serializacji do zapamiętania jakiegoś stanu programu i przywrócenia go do tego stanu później wygląda całkiem przekonywująco. Jednak zanim zaczniemy ją stosować, wypadałoby odpowiedzieć na kilka pytań. Co stanie się przy serializacji dwóch obiektów, jeśli oba zawierają referencję do trzeciego? Kiedy się je odtworzy, czy otrzymamy tylko jedno wystąpienie trzeciego obiektu? A co będzie, jeśli dokonamy serializacji tych obiektów do dwóch różnych plików, a deserializacji w różnych częściach kodu?

Poniższy przykład pokazuje ten problem:

```
//: io/MyWorld.java
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class House implements Serializable {}

class Animal implements Serializable {
  private String name;
  private House preferredHouse;
  Animal(String nm, House h) {
    name = nm;
    preferredHouse = h;
```

```
  }
  public String toString() {
    return name + "[" + super.toString() +
      "], " + preferredHouse + "\n";
  }
}

public class MyWorld {
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    House house = new House();
    List<Animal> animals = new ArrayList<Animal>();
    animals.add(new Animal("Kundel Bosco", house));
    animals.add(new Animal("Chomik Reks", house));
    animals.add(new Animal("Kotka Molly", house));
    print("animals: " + animals);
    ByteArrayOutputStream buf1 =
      new ByteArrayOutputStream();
    ObjectOutputStream o1 = new ObjectOutputStream(buf1);
    o1.writeObject(animals);
    o1.writeObject(animals); // Zapis drugiego zestawu
    // Zapis do innego strumienia:
    ByteArrayOutputStream buf2 =
      new ByteArrayOutputStream();
    ObjectOutputStream o2 = new ObjectOutputStream(buf2);
    o2.writeObject(animals);
    // A teraz odtwarzanie:
    ObjectInputStream in1 = new ObjectInputStream(
      new ByteArrayInputStream(buf1.toByteArray()));
    ObjectInputStream in2 = new ObjectInputStream(
      new ByteArrayInputStream(buf2.toByteArray()));
    List
      animals1 = (List)in1.readObject(),
      animals2 = (List)in1.readObject(),
      animals3 = (List)in2.readObject();
    print("animals1: " + animals1);
    print("animals2: " + animals2);
    print("animals3: " + animals3);
  }
} /* Output: (Sample)
animals: [Kundel Bosco[Animal@addbf1], House@42e816
, Chomik Reks[Animal@9304b1], House@42e816
, Kotka Molly[Animal@190d11], House@42e816
]
animals1: [Kundel Bosco[Animal@de6f34], House@156ee8e
, Chomik Reks[Animal@47b480], House@156ee8e
, Kotka Molly[Animal@19b49e6], House@156ee8e
]
animals2: [Kundel Bosco[Animal@de6f34], House@156ee8e
, Chomik Reks[Animal@47b480], House@156ee8e
, Kotka Molly[Animal@19b49e6], House@156ee8e
]
animals3: [Kundel Bosco[Animal@10d448], House@e0e1c6
, Chomik Reks[Animal@6ca1c], House@e0e1c6
, Kotka Molly[Animal@1bf216a], House@e0e1c6
]
*///:~
```

Pierwszą interesującą rzeczą jest fakt, że można wykorzystać serializację do i z tablicy bajtów jako sposób wykonania „głębokiej kopii" dowolnego obiektu `Serializable` (określamy ją jako „głęboką", ponieważ duplikuje się całą sieć obiektów, nie zaś tylko pierwotny obiekt i zawarte w nim referencje). Kopiowanie zostało dokładnie omówione w dostępnych w Internecie suplementach towarzyszących książce.

Obiekty `Animal` zawierają pola typu `House`. W `main()` tworzona jest lista `ArrayList` obiektów `Animal` i serializowana dwukrotnie do jednego strumienia, a następnie jeszcze raz do drugiego. Kiedy są one deserializowane i wypisane, zobaczymy je na wyjściu programu (obiekty za każdym przebiegiem będą w innych miejscach pamięci).

Oczywiście oczekiwaliśmy, że deserializowane obiekty będą miały inne adresy niż oryginalne. Zauważmy jednak, że w `animals1` i `animals2` pojawiają się te same adresy, łącznie z referencją do wspólnego dla nich obiektu `House`. Z drugiej strony, kiedy odtwarzana jest lista `animals3`, system nie ma możliwości zorientowania się, że obiekty w drugim i w pierwszym strumieniu są w rzeczywistości tymi samymi obiektami, zatem tworzy różną, nową sieć obiektów.

Dopóki zapisuje się wszystkie serializowane obiekty do jednego strumienia, istnieje możliwość odtworzenia oryginalnej sieci obiektów bez przypadkowego ich duplikowania. Oczywiście da się zmienić stan obiektów w czasie pomiędzy zapisaniem pierwszego a ostatniego z nich, ale czyni się to na własną odpowiedzialność — obiekty zostaną zapisane dokładnie w takim stanie (i z takimi połączeniami do innych), w jakim znajdują się w momencie serializacji.

Najbezpieczniejszym rozwiązaniem, gdy chce się zapisać stan całego systemu, jest dokonywanie serializacji jako operacji „atomowej" (niepodzielnej). Jeżeli serializujemy grupę obiektów, wykonujemy jakieś inne działania, serializujemy następną grupę itd., zapis nie będzie wiarygodny. Zamiast tego należy umieścić wszystkie obiekty, które mają wpływ na stan naszego systemu, w pojedynczym kontenerze i po prostu zapisać ten kontener za pomocą jednej operacji. Można go potem odtworzyć również pojedynczym wywołaniem metody.

Następny przykład to wyimaginowany system CAD (Computer-Aided Design, projektowanie wspomagane komputerowo), który demonstruje właśnie takie rozwiązanie. Oprócz tego mamy tu pole typu `static` — według dokumentacji klasa `Class` implementuje interfejs `Serializable`, zapisanie pola `static` powinno być więc łatwo osiągalne przez serializowanie obiektu `Class` odpowiedniego dla klasy. W każdym razie wygląda to na sensowne wyjście.

```
//: io/StoreCADState.java
// Utrwalanie stanu wyimaginowanego systemu CAD.
import java.io.*;
import java.util.*;

abstract class Shape implements Serializable {
  public static final int RED = 1, BLUE = 2, GREEN = 3;
  private int xPos, yPos, dimension;
  private static Random rand = new Random(47);
  private static int counter = 0;
  public abstract void setColor(int newColor);
  public abstract int getColor();
```

```
  public Shape(int xVal, int yVal, int dim) {
    xPos = xVal;
    yPos = yVal;
    dimension = dim;
  }
  public String toString() {
    return getClass() +
      "color[" + getColor() + "] xPos[" + xPos +
      "] yPos[" + yPos + "] dim[" + dimension + "]\n";
  }
  public static Shape randomFactory() {
    int xVal = rand.nextInt(100);
    int yVal = rand.nextInt(100);
    int dim = rand.nextInt(100);
    switch(counter++ % 3) {
      default:
      case 0: return new Circle(xVal, yVal, dim);
      case 1: return new Square(xVal, yVal, dim);
      case 2: return new Line(xVal, yVal, dim);
    }
  }
}

class Circle extends Shape {
  private static int color = RED;
  public Circle(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void setColor(int newColor) { color = newColor; }
  public int getColor() { return color; }
}

class Square extends Shape {
  private static int color;
  public Square(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
    color = RED;
  }
  public void setColor(int newColor) { color = newColor; }
  public int getColor() { return color; }
}

class Line extends Shape {
  private static int color = RED;
  public static void
  serializeStaticState(ObjectOutputStream os)
  throws IOException { os.writeInt(color); }
  public static void
  deserializeStaticState(ObjectInputStream os)
  throws IOException { color = os.readInt(); }
  public Line(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void setColor(int newColor) { color = newColor; }
  public int getColor() { return color; }
}
```

```
public class StoreCADState {
  public static void main(String[] args) throws Exception {
    List<Class<? extends Shape>> shapeTypes =
      new ArrayList<Class<? extends Shape>>();
    // Dodanie referencji do obiektów Class poszczególnych klas:
    shapeTypes.add(Circle.class);
    shapeTypes.add(Square.class);
    shapeTypes.add(Line.class);
    List<Shape> shapes = new ArrayList<Shape>();
    // Utworzenie kilku figur:
    for(int i = 0; i < 10; i++)
      shapes.add(Shape.randomFactory());
    // Ustawienie statycznych składowych koloru na GREEN:
    for(int i = 0; i < 10; i++)
      ((Shape)shapes.get(i)).setColor(Shape.GREEN);
    // Zapis tablicy stanu:
    ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream("CADState.out"));
    out.writeObject(shapeTypes);
    Line.serializeStaticState(out);
    out.writeObject(shapes);
    // Wypisanie figur:
    System.out.println(shapes);
  }
} /* Output:
[class Circlecolor[3] xPos[58] yPos[55] dim[93]
, class Squarecolor[3] xPos[61] yPos[61] dim[29]
, class Linecolor[3] xPos[68] yPos[0] dim[22]
, class Circlecolor[3] xPos[7] yPos[88] dim[28]
, class Squarecolor[3] xPos[51] yPos[89] dim[9]
, class Linecolor[3] xPos[78] yPos[98] dim[61]
, class Circlecolor[3] xPos[20] yPos[58] dim[16]
, class Squarecolor[3] xPos[40] yPos[11] dim[22]
, class Linecolor[3] xPos[4] yPos[83] dim[6]
, class Circlecolor[3] xPos[75] yPos[10] dim[42]
]
*///:~
```

Klasa Shape (figura) implementuje Serializable, tak więc wszystkie klasy dziedziczące po Shape będą automatycznie również serializowalne. Każdy obiekt Shape zawiera dane, a każda klasa pochodna od Shape ma pole static, które decyduje o kolorze wszystkich figur danego podtypu (umieszczenie pola static w klasie bazowej skutkowałoby stworzeniem tylko jednego pola wspólnego dla całej hierarchii, ponieważ pola statyczne nie są duplikowane w klasach pochodnych). Metody w klasie bazowej mogą zostać przesłonięte, aby ustawiać kolory dla różnych typów (metody static nie są wiązane dynamicznie, więc są to zwykłe metody). Metoda randomFactory() przy każdym wywołaniu wytwarza inny obiekt Shape, wykorzystując losowe wartości dla danych Shape.

Klasy Circle i Square są prostymi rozszerzeniami klasy Shape; jedyną różnicą jest to, że Circle inicjalizuje zmienną Color w miejscu definicji, a Square wewnątrz konstruktora. Klasę Line omówimy później.

W main() używane są dwie listy ArrayList: pierwsza przechowuje obiekty Class, a druga — figury.

Odtwarzanie obiektów jest zupełnie nieskomplikowane:

```
//: io/RecoverCADState.java
// Odtwarzanie stanu wyimaginowanego systemu CAD.
// {RunFirst: StoreCADState}
import java.io.*;
import java.util.*;

public class RecoverCADState {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("CADState.out"));
    // Wczytywanie w kolejności zgodnej z kolejnością zapisu:
    List<Class<? extends Shape>> shapeTypes =
      (List<Class<? extends Shape>>)in.readObject();
    Line.deserializeStaticState(in);
    List<Shape> shapes = (List<Shape>)in.readObject();
    System.out.println(shapes);
  }
} /* Output:
[class Circlecolor[1] xPos[58] yPos[55] dim[93]
, class Squarecolor[0] xPos[61] yPos[61] dim[29]
, class Linecolor[3] xPos[68] yPos[0] dim[22]
, class Circlecolor[1] xPos[7] yPos[88] dim[28]
, class Squarecolor[0] xPos[51] yPos[89] dim[9]
, class Linecolor[3] xPos[78] yPos[98] dim[61]
, class Circlecolor[1] xPos[20] yPos[58] dim[16]
, class Squarecolor[0] xPos[40] yPos[11] dim[22]
, class Linecolor[3] xPos[4] yPos[83] dim[6]
, class Circlecolor[1] xPos[75] yPos[10] dim[42]
]
*///:~
```

Jak widać, wartości `xPos`, `yPos` i `dim` zostały zapisane i odtworzone poprawnie, jednak coś jest nie w porządku z odczytaniem wartości zmiennej statycznej. Przy zapisywaniu wszędzie jest „3", jednak po odzyskaniu wartość nie jest już ta sama. Obiekty `Circle` mają wartość 1 (`RED`, tak jak zostały zdefiniowane), natomiast `Square` mają wartość 0 (jak pamiętamy, była ona inicjalizowana w konstruktorze). Wygląda na to, że pola statyczne nie były w ogóle serializowane! To prawda — mimo że klasa `Class` implementuje interfejs `Serializable`, implementacja ta nie działa zgodnie z naszymi oczekiwaniami. Zatem jeśli chcemy serializować pola statyczne, musimy się o to zatroszczyć sami.

Właśnie do tego służą metody statyczne `serializeStaticState()` i `deserializeStaticState()`, dodane w klasie `Line`. Widać, że są one wywoływane jawnie jako część procesu zapamiętywania i odzyskiwania danych (kolejność zapisywania do pliku i odczytywania z powrotem musi zostać utrzymana). Aby zatem programy zadziałały poprawnie, należy:

1. Dodać metodę `serializeStaticState()` i `deserializeStaticState()` do podklas `Shape`.
2. Usunąć listę `shapeTypes` i cały kod z nią związany.
3. Dodać wywołania nowych metod statycznych dla wszystkich figur.

Trzeba również pomyśleć o bezpieczeństwie — serializacja obejmuje także dane prywatne (`private`). Jeżeli istnieją jakieś wymagania bezpieczeństwa, pola te powinny być oznaczone jako `transient`. Wtedy jednak należy zaprojektować bezpieczny sposób przechowania informacji, tak aby można było potem odtworzyć stan zmiennych prywatnych.

Ćwiczenie 30. Popraw programy *StoreCADState.java* i *RestoreCADState.java* w sposób opisany w tekście (1).

XML

Istotnym ograniczeniem techniki serializacji jest uzależnienie rozwiązania utrwalania obiektów od platformy: deserializacja tak utrwalonych obiektów jest możliwa tylko z poziomu programów języka Java. Gdyby zależało nam na rozwiązaniu międzyplatformowym, moglibyśmy skonwertować dane o obiektach na format XML, który jest bezproblemowo przetwarzany na większości platform systemowych i programowych.

Popularność języka XML owocuje nadmiarem opcji programistycznych; programista Javy ma między innymi do dyspozycji klasy biblioteki `javax.xml.*` wchodzące w skład zestawu JDK. Do omówienia wybrałem jednak otwartą bibliotekę XOM (napisał ją Elliotte Rusty Harold — jej dokumentacja publikowana jest pod adresem *www.xom.nu*, stamtąd można też pobrać bibliotekę); zdaje się, że to najprostsza i najmniej kłopotliwa metoda tworzenia i modyfikowania dokumentów XML w Javie. Do tego XOM kładzie nacisk na poprawność generowanych dokumentów XML.

W ramach przykładu załóżmy, że dysponujemy obiektami klasy `Person` zawierającymi imiona i nazwiska osób; dane te chcemy utrwalić w pliku XML. Prezentowana poniżej klasa `Person` posiada metodę `getXML()`, która za pomocą XOM konwertuje dane obiektu `Person` na element XML (obiekt klasy `Element`); z kolei konstruktor klasy przyjmuje obiekt `Element` i wyłuskuje z niego dane odpowiednie dla obiektu klasy `Person` (pliki źródłowe przykładów dotyczących języka XML zostały przeniesione do osobnego katalogu):

```
//: xml/Person.java
// Zastosowanie biblioteki XOM do zapisu i odczytu dokumentu XML
// {Requires: nu.xom.Node; Wymaga zainstalowania biblioteki XOM
// spod adresu http://www.xom.nu }
import nu.xom.*;
import java.io.*;
import java.util.*;

public class Person {
  private String first, last;
  public Person(String first, String last) {
    this.first = first;
    this.last = last;
  }
  // Tworzy element (Element) XML na bazie obiektu Person (this):
  public Element getXML() {
    Element person = new Element("person");
    Element firstName = new Element("first");
```

```
    firstName.appendChild(first);
    Element lastName = new Element("last");
    lastName.appendChild(last);
    person.appendChild(firstName);
    person.appendChild(lastName);
    return person;
  }
  // Konstruktor przywracający obiekt Person z elementu XML:
  public Person(Element person) {
    first= person.getFirstChildElement("first").getValue();
    last = person.getFirstChildElement("last").getValue();
  }
  public String toString() { return first + " " + last; }
  // Dla czytelności:
  public static void
  format(OutputStream os, Document doc) throws Exception {
    Serializer serializer= new Serializer(os,"Windows-1250");
    serializer.setIndent(4);
    serializer.setMaxLength(60);
    serializer.write(doc);
    serializer.flush();
  }
  public static void main(String[] args) throws Exception {
    List<Person> people = Arrays.asList(
      new Person("Prof. Ambroży", "Kleks"),
      new Person("Alojzy", "Bąbel"),
      new Person("Max", "Benson"));
    System.out.println(people);
    Element root = new Element("people");
    for(Person p : people)
      root.appendChild(p.getXML());
    Document doc = new Document(root);
    format(System.out, doc);
    format(new BufferedOutputStream(new FileOutputStream(
      "People.xml")), doc);
  }
} /* Output:
[Prof. Ambroży Kleks, Alojzy Bąbel, Max Benson]
<?xml version="1.0" encoding="Windows-1250"?>
<people>
  <person>
    <first>Prof. Ambroży</first>
    <last>Kleks</last>
  </person>
  <person>
    <first>Alojzy</first>
    <last>Bąbel</last>
  </person>
  <person>
    <first>Max</first>
    <last>Benson</last>
  </person>
</people>
*///:~
```

Działanie metod biblioteki XOM nie wymaga szerszego komentarza — a jeśli jednak, to można go znaleźć w dokumentacji XOM.

Biblioteka XOM zawiera klasę `Serializer`, którą wykorzystujemy do wywołania metody `format()` w celu zapisania dokumentu XML w postaci czytelniejszej dla człowieka (z wcięciami i innymi elementami formatowania). `Serializer` to narzędzie o tyle pożyteczne, że gdybyśmy ograniczyli się do wywołania `toXML()`, otrzymalibyśmy dokument XML pisany „jednym ciągiem".

Deserializacja obiektów klasy `Person` z pliku XML również nie powinna przysparzać trudności:

```
//: xml/People.java
// {Requires: nu.xom.Node; Wymaga zainstalowania biblioteki XOM
// spod adresu http://www.xom.nu }
// {RunFirst: Person}
import nu.xom.*;
import java.util.*;

public class People extends ArrayList<Person> {
  public People(String fileName) throws Exception {
    Document doc = new Builder().build(fileName);
    Elements elements =
      doc.getRootElement().getChildElements();
    for(int i = 0; i < elements.size(); i++)
      add(new Person(elements.get(i)));
  }
  public static void main(String[] args) throws Exception {
    People p = new People("People.xml");
    System.out.println(p);
  }
} /* Output:
[Prof. Ambroży Kleks, Alojzy Bąbel, Max Benson]
*///:~
```

Konstruktor klasy `People` otwiera i wczytuje dokument XML za pośrednictwem metody `Builder.build()` z pakietu XOM; późniejsze wywołanie `getChildElements()` zwraca listę `Elements` (nie jest to standardowy kontener Javy, ale obiekt wyposażony jedynie w metody `get()` i `size()` — Harold, twórca biblioteki, nie chciał zmuszać programistów do instalowania Javy SE5, zaimplementował więc szczątkowo własną klasę kontenera). Każdy element tej listy (obiekt typu `Element`) reprezentuje pojedynczy obiekt `Person` (w postaci elementu XML), więc może zostać przekazany do konstruktora klasy `Person`. Takie odtwarzanie wymaga znajomości struktury pliku XML, ale w tego rodzaju zadaniach jest to dość typowe. Jeśli struktura dokumentu jest inna od oczekiwanej, XOM zgłosi wyjątek. Alternatywą jest rozbudowanie kodu tak, aby najpierw badał strukturę dokumentu XML — w ten sposób można by zrównoważyć niepełne informacje o otrzymywanych dokumentach.

Aby oba przykłady dały się skompilować, należy w zasięgu ścieżki `CLASSPATH` umieścić pliki JAR z biblioteką XOM.

To siłą rzeczy pobieżne wprowadzenie do programowania XML w Javie z użyciem biblioteki XOM; zainteresowanych pogłębieniem wiedzy w tym zakresie odsyłam pod adres *www.xom.nu*.

Ćwiczenie 31. Dodaj do klas `Person` i `People` pola adresowe (2).

Ćwiczenie 32. Napisz program, który za pomocą klas `Map<String,Integer>` i `net.mindview.util.TextFile` zliczy wystąpienia słów w pliku (drugim argumentem wywołania konstruktora `TextFile` powinno być wyrażenie „`\\W+`"). Zapisz wynik zliczania w pliku XML (4).

Preferencje

W JDK 1.4 wprowadzono interfejs programistyczny preferencji (*Preferences API*), który o wiele bardziej odpowiada idei trwałości obiektów, gdyż pozwala na automatyczne zapisywanie i odtwarzanie informacji. Jednak interfejs ten można wykorzystywać wyłącznie do obsługi niewielkich i ograniczonych zbiorów danych — pozwala on wyłącznie na przechowywanie danych typów podstawowych i ciągów znaków, przy czym wielkość tych ostatnich nie może przekraczać 8 kilobajtów (nie jest to mało, ale i tak nie będziesz chyba próbował tworzyć czegokolwiek poważnego, dysponując takimi możliwościami). Zgodnie z tym, co sugeruje nazwa, API służy do przechowywania i pobierania preferencji użytkownika oraz ustawień konfiguracyjnych programów.

Preferencje są zbiorami par klucz-wartość (podobnie jak obiekty `Map`) przechowywanymi w hierarchii węzłów. Choć korzystając z tej hierarchii, można tworzyć skomplikowane struktury, to zazwyczaj jednak tworzony jest jeden węzeł o nazwie odpowiadającej nazwie klasy i w nim są przechowywane wszystkie informacje. Oto prosty przykład:

```
//: io/PreferencesDemo.java
import java.util.prefs.*;
import static net.mindview.util.Print.*;

public class PreferencesDemo {
  public static void main(String[] args) throws Exception {
    Preferences prefs = Preferences
      .userNodeForPackage(PreferencesDemo.class);
    prefs.put("Miejsce akcji", "Oz");
    prefs.put("Obuwie", "czerwone pantofelki");
    prefs.putInt("Kamraci", 4);
    prefs.putBoolean("Jakieś wiedźmy?", true);
    int usageCount = prefs.getInt("UsageCount", 0);
    usageCount++;
    prefs.putInt("Licznik", usageCount);
    for(String key : prefs.keys())
      print(key + ": "+ prefs.get(key, null));
    // Koniecznie podać wartość domyślną:
    print("Ilu towarzyszy podróży miała Dorotka? " +
      prefs.getInt("Kamraci", 0));
  }
} /* Output: (Sample)
Miejsce akcji: Oz
Obuwie: czerwone pantofelki
Kamraci: 4
Jakieś wiedźmy?: true
Licznik: 53
Ilu towarzyszy podróży miała Dorotka? 4
*///:~
```

W powyższym programie została wykorzystana metoda `userNodeForPackage()`, równie dobrze można jednak użyć metody `systemNodeForPackage()`; wybór jest tu dosyć dowolny, jednak ogólna idea jest taka, że ustawienia użytkownika („user") są wykorzystywane do przechowywania preferencji poszczególnych użytkowników, a ustawienia systemowe („system") — konfiguracji ogólnej. Ponieważ metoda `main()` jest statyczna, zatem do identyfikacji węzła wykorzystywana jest klasa `PreferencesDemo.class`; w metodach, które nie są statyczne, zazwyczaj wykorzystywany będzie obiekt zwrócony przez metodę `getClass()`. Do identyfikacji węzła nie trzeba używać bieżącej klasy, choć jest to standardowo stosowane rozwiązanie.

Po utworzeniu węzła można go używać zarówno do zapisu, jak i odczytu danych. W powyższym przykładzie w węźle zapisywane są różne typy danych, a następnie zostają pobrane ich klucze (przy użyciu metody `keys()`). Metoda ta zwraca tablicę ciągów znaków, co może być nieco niespodziewane, zwłaszcza jeśli jesteśmy przyzwyczajeni do analogicznych metod dostępnych w bibliotece kolekcji. Warto zwrócić uwagę na drugi argument metody `get()`. Określa on wartość domyślną, zwracaną gdy nie ma żadnych informacji skojarzonych z danym kluczem. Operując na zbiorze kluczy, można mieć pewność, że zawsze będą z nimi skojarzone wartości; dlatego też użycie `null` jako wartości drugiego argumentu metody `get()` jest bezpieczne. Jednak zazwyczaj wartość klucza będzie pobierana w następujący sposób:

```
prefs.getInt("Kamraci", 0);
```

W normalnych przypadkach będziemy starali się podać sensowną wartość domyślną. Następujący kod przedstawia stosowane zwykle rozwiązanie:

```
int usageCount = prefs.getInt("Licznik", 0);
usageCount++;
prefs.putInt("Licznik", usageCount);
```

W ten sposób podczas pierwszego uruchomienia programu wartość `Licznik` będzie pusta, lecz we wszystkich kolejnych przypadkach wartość ta będzie już większa od zera.

Testując program *PreferencesDemo.java*, można się przekonać, że każde jego uruchomienie inkrementuje wartość preferencji `usageCount`. Gdzie jednak te informacje są przechowywane? Nie ma żadnego pliku lokalnego, który pojawiłby się po pierwszym uruchomieniu programu. Otóż wykonując swoje zadania, interfejs programistyczny preferencji wykorzystuje odpowiednie zasoby systemowe, zatem sposób przechowywania danych będzie zależny od używanego systemu operacyjnego. W systemach Windows do tego celu jest wykorzystywany rejestr systemowy (gdyż z założenia jest to hierarchia węzłów zawierających pary nazwa-wartość). Niemniej jednak najistotniejszy jest fakt, że informacje zostają w jakiś magiczny sposób zapisane, dzięki czemu programista nie musi przejmować się szczegółami rozwiązania w poszczególnych systemach operacyjnych.

To jedynie niewielka część informacji dotyczących interfejsu programistycznego do obsługi preferencji, więcej szczegółów można znaleźć w dokumentacji JDK.

Ćwiczenie 33. Napisz program, który wypisze bieżącą wartość preferencji „katalog bazowy" i będzie pytał użytkownika o nową wartość preferencji. Wykorzystaj interfejs Preferences API do zapisania podanej wartości (2).

Podsumowanie

Biblioteka strumieni wejścia-wyjścia Javy spełnia podstawowe wymagania: pozwala na odczytywanie i zapisywanie danych na konsolę, do pliku, bloku pamięci, a nawet przez internet. Przez dziedziczenie można tworzyć nowe typy obiektów wejścia i wyjścia. W prosty sposób można także rozszerzyć zbiór typów obiektów akceptowanych przez strumień, przedefiniowując metodę `toString()`, która jest wywoływana automatycznie, kiedy przekazuje się obiekt metodzie oczekującej jako argumentu ciągu znaków (ograniczona „automatyczna konwersja typów" Javy).

Są jednak pytania pozostawione bez odpowiedzi przez dokumentację i sposób zaprojektowania biblioteki strumieni wejścia-wyjścia. Na przykład byłoby miło, gdyby można było zażądać zgłoszenia wyjątku przy próbie nadpisania pliku otwartego jako wyjście — niektóre systemy programowania pozwalają na zastrzeżenie, że chcemy otworzyć plik do zapisu tylko, jeśli do tej pory nie istniał. Wydaje się, że w Javie najpierw należy użyć obiektu `File`, aby stwierdzić, czy plik istnieje, ponieważ jeśli otworzymy go jako `FileOutputStream` albo `FileWriter`, zawsze zostanie nadpisany.

Biblioteka strumieniowa wejścia-wyjścia wywołuje mieszane uczucia: realizuje sporo zadań i jest niezależna od platformy. Jednak jeśli nie rozumie się wzorca projektowego „dekoratora", jej budowa jest nieintuicyjna, co stanowi dodatkową trudność w jej objaśnianiu i nauce. Jest również niekompletna; na przykład powinny w niej istnieć klasy analogiczne do klasy `TextFile`, którą musiałem tworzyć sam (dostępna w Javie SE5 klasa `PrintWriter` stanowi krok naprzód, ale to tylko rozwiązanie częściowe). Edycja SE5 załatała wiele dziur: wreszcie doczekaliśmy się formatowania wyjścia, które w większości innych języków dostępne było od zawsze.

Kiedy już *ogarniesz* ideę dekoratora i zaczniesz używać biblioteki w sytuacjach wymagających elastyczności, jaką on zapewnia, możesz czerpać korzyści z takiego sposobu jej zaprojektowania, mimo kosztów w postaci dodatkowych wierszy kodu.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 19.

Typy wyliczeniowe

Słowo kluczowe `enum` *pozwala na tworzenie nowego typu z ograniczonym zestawem wartości nazwanych i traktowanie tych wartości jako zwykłych komponentów programu. To wielce przydatna możliwość*[1].

Typy wyliczeniowe (bo tak będziemy je nazywać) zostały już pobieżnie zaprezentowane w rozdziale „Inicjalizacja i sprzątanie". Teraz, po zaprezentowaniu kilku bardziej zaawansowanych aspektów Javy, przyjrzymy się bliżej mechanizmowi wyliczeń w Javie SE5. Przekonasz się, że typy wyliczeniowe można wykorzystać w interesujący sposób, ale i dowiesz się więcej o pozostałych cechach języka, w tym o refleksji i typach ogólnych.

Podstawowe cechy typów wyliczeniowych

Z rozdziału „Inicjalizacja i sprzątanie" wiesz, że listę stałych wyliczeniowych można przeglądać wywołaniami metody `values()` wyliczenia. Metoda `values()` zwraca tablicę stałych wyliczenia ułożonych w kolejności, w jakiej były deklarowane; otrzymaną tablicę można przeglądać choćby w pętli foreach.

Utworzenie typu wyliczeniowego oznacza wygenerowanie przez kompilator odpowiadającej mu klasy. Klasa ta jest automatycznie wyprowadzana z klasy `java.lang.Enum`, której cechy ilustruje poniższy przykład:

```
//: enumerated/EnumClass.java
// Cechy klasy Enum
import static net.mindview.util.Print.*;

enum Shrubbery { GROUND, CRAWLING, HANGING }

public class EnumClass {
  public static void main(String[] args) {
    for(Shrubbery s : Shrubbery.values()) {
```

[1] W opracowaniu materiału do tego rozdziału nieocenionej pomocy udzielił mi Joshua Bloch.

```
      print(s + " liczba porządkowa: " + s.ordinal());
      printnb(s.compareTo(Shrubbery.CRAWLING) + " ");
      printnb(s.equals(Shrubbery.CRAWLING) + " ");
      print(s == Shrubbery.CRAWLING);
      print(s.getDeclaringClass());
      print(s.name());
      print("----------------------");
    }
    // Wygenerowanie wartości wyliczenia na podstawie ciągu z nazwą:
    for(String s : "HANGING CRAWLING GROUND".split(" ")) {
      Shrubbery shrub = Enum.valueOf(Shrubbery.class, s);
      print(shrub);
    }
  }
} /* Output:
GROUND liczba porządkowa: 0
-1 false false
class Shrubbery
GROUND
----------------------
CRAWLING liczba porządkowa: 1
0 true true
class Shrubbery
CRAWLING
----------------------
HANGING liczba porządkowa: 2
1 false false
class Shrubbery
HANGING
----------------------
HANGING
CRAWLING
GROUND
*///:~
```

Metoda `ordinal()` zwraca wartość typu `int` wskazującą kolejność wystąpienia danej stałej w deklaracji typu wyliczeniowego (pozycje są liczone od zera). Składowe danego typu wyliczeniowego można zawsze bezpiecznie porównywać za pośrednictwem operatora == oraz automatycznie syntetyzowanych metod `equals()` i `hashCode()`. Klasa `Enum` implementuje interfejs `Comparable`, posiada więc również metodę `compareTo()`. Do tego wyliczenia implementują interfejs `Serializable`.

Jeśli na rzecz egzemplarza wyliczenia wywołasz metodę `getDeclaringClass()`, otrzymasz referencję klasy wyliczenia.

Metoda `name()` zwraca ciąg nazwy ściśle zgodny z tym występującym w deklaracji; ten sam ciąg zwracany jest przez wywołania `toString()`. Metoda `valueOf()` to statyczna składowa `Enum` zwracająca egzemplarz wyliczenia odpowiadający ciągowi przekazanemu w wywołaniu; w przypadku braku podanej nazwy w wyliczeniu wywołanie spowoduje zgłoszenie wyjątku.

Wyliczenia a importy statyczne

Przeanalizujmy modyfikację przykładu *Burito.java* z rozdziału „Inicjalizacja i sprzątanie”:

```
//: enumerated/Spiciness.java
package enumerated;

public enum Spiciness {
  NIJAKIE, ŁAGODNE, PIKANTNE, OSTRE, PALĄCE
} ///:~
//: enumerated/Burrito.java
package enumerated;
import static enumerated.Spiciness.*;

public class Burrito {
  Spiciness degree;
  public Burrito(Spiciness degree) { this.degree = degree;}
  public String toString() { return "To burrito jest "+ degree;}
  public static void main(String[] args) {
    System.out.println(new Burrito(ŁAGODNE));
    System.out.println(new Burrito(PIKANTNE));
    System.out.println(new Burrito(OSTRE));
  }
} /* Output:
To burrito jest ŁAGODNE
To burrito jest PIKANTNE
To burrito jest OSTRE
*///:~
```

Statyczny import wciąga wszystkie identyfikatory typu wyliczeniowego do lokalnej przestrzeni nazw, nie trzeba więc ich kwalifikować nazwą typu wyliczeniowego. Czy to dobry pomysł czy lepiej jawnie kwalifikować wszystkie egzemplarze wyliczeń? Odpowiedź zależy zapewne od stopnia złożoności kodu. Kompilator z pewnością nie pozwoli na zastosowanie niepoprawnego typu, więc cała kwestia sprowadza się do czytelności kodu. W wielu sytuacjach import statyczny będzie sprawdzał się świetnie, trzeba to jednak rozpatrywać dla każdego programu indywidualnie.

Zaprezentowanej techniki nie można zastosować, jeśli typ wyliczeniowy jest definiowany w tym samym pliku albo w pakiecie domyślnym (najwyraźniej jest to wynik jakiejś dyskusji w łonie firmy Sun).

Dodawanie metod do typów wyliczeniowych

Poza tym, że po typie wyliczeniowym nie można dziedziczyć, zachowuje się on jak najzwyklejsza klasa. Oznacza to, że można w nim definiować własne metody. Ba, typ wyliczeniowy można nawet wyposażyć w metodę `main()`.

Niekiedy zachodzi potrzeba wygenerowania dla typu wyliczeniowego alternatywnej reprezentacji tekstowej, bo ta zwracana przez `toString()` ogranicza się do nazwy egzemplarza wyliczenia, o czym mieliśmy już okazję się przekonać. Aby osiągnąć swój cel, powinniśmy udostępnić w wyliczeniu konstruktor przyjmujący dodatkowe informacje i metodę zwracającą rozszerzony opis wyliczenia, jak tu:

```
//: enumerated/OzWitch.java
// Wiedźmy w krainie Oz.
import static net.mindview.util.Print.*;

public enum OzWitch {
  // Nazwy typu wyliczeniowego muszą być zdefiniowane przed metodami:
  ZACHÓD("Pani Gulch, vel Zła Czarownica z Zachodu"),
  PÓŁNOC("Glinda, Dobra Wróżka z Północy"),
  WSCHÓD("Zła Czarownica ze Wschodu, właścicielka czerwonych pantofelków"),
  POŁUDNIE("Dobra, ale zaginiona");
  private String description;
  // Konstruktor powinien mieć dostęp prywatny albo pakietowy:
  private OzWitch(String description) {
    this.description = description;
  }
  public String getDescription() { return description; }
  public static void main(String[] args) {
    for(OzWitch witch : OzWitch.values())
      print(witch + ": " + witch.getDescription());
  }
} /* Output:
ZACHÓD: Pani Gulch, vel Zła Czarownica z Zachodu
PÓŁNOC: Glinda, Dobra Wróżka z Północy
WSCHÓD: Zła Czarownica ze Wschodu, właścicielka czerwonych pantofelków
POŁUDNIE: Dobra, ale zaginiona
*///:~
```

Zauważ, że jeśli typ wyliczeniowy ma zawierać definicje metod, to lista nazw musi się kończyć średnikiem. Do tego Java wymusza, aby nazwy były pierwszymi elementami definicji typu wyliczeniowego. Próba definiowania nazw za polami czy metodami spowoduje błąd kompilacji.

Konstruktor i metody mają postać jak w zwyczajnych klasach, bo poza paroma ograniczeniami typ wyliczeniowy *jest* zwyczajną klasą. Można z nim zrobić niemal wszystko, w granicach rozsądku oczywiście.

Choć konstruktor typu wyliczeniowego w prezentowanym przykładzie jest prywatny, to tryb dostępu jest w zasadzie nieistotny — konstruktor i tak może być wykorzystywany jedynie do tworzenia egzemplarzy typu wyliczeniowego deklarowanych wewnątrz definicji tego typu; kompilator nie pozwoli na tworzenie żadnych nowych nazw po zakończeniu definicji typu wyliczeniowego.

Przesłanianie metod typu wyliczeniowego

Jest jeszcze jeden sposób na wygenerowanie alternatywnych ciągów dla poszczególnych nazw typu wyliczeniowego. W poniższym przykładzie same nazwy uznajemy za odpowiednie, ale ich pisownia (tradycyjna, z wielkimi literami) jest nieodpowiednia dla komunikatów wyjściowych. Przesłaniamy więc metodę `toString()` typu wyliczeniowego, tak jak zrobilibyśmy to w każdej zwyczajnej klasie:

```
//: enumerated/SpaceShip.java
public enum SpaceShip {
  ZWIADOWCZY, PROM, TRANSPORTOWIEC, KRĄŻOWNIK, NISZCZYCIEL, BAZA;
  public String toString() {
```

```
    String id = name();
    String lower = id.substring(1).toLowerCase();
    return id.charAt(0) + lower;
  }
  public static void main(String[] args) {
    for(SpaceShip s : values()) {
      System.out.println(s);
    }
  }
} /* Output:
Zwiadowczy
Prom
Transportowiec
Krążownik
Niszczyciel
Baza
*///:~
```

Metoda `toString()` pobiera nazwę z wyliczenia `SpaceShip` wywołaniem `name()` i modyfikuje otrzymany ciąg tak, aby tylko pierwsza litera była wielka.

Wyliczenia w instrukcjach wyboru

Bardzo wygodną cechą wyliczeń jest możliwość używania ich w instrukcjach wyboru `switch`. Normalnie instrukcja `switch` działa jedynie z wartościami całkowitymi, ale nazwy są w typach wyliczeniowych skojarzone właśnie z liczbami porządkowymi, a kolejność tych nazw w wyliczeniu można ustalić wywołaniem metody `ordinal()`. Najwyraźniej kompilator posługuje się podobną techniką, w każdym razie wyliczenia można śmiało stosować w instrukcjach `switch`.

Choć normalnie nazwę wyliczenia trzeba kwalifikować nazwą typu, w klauzulach `case` można z tego zrezygnować. Oto przykład wykorzystujący typ wyliczeniowy do utworzenia prostego automatu stanów:

```
//: enumerated/TrafficLight.java
// Typy wyliczeniowe w instrukcjach wyboru.
import static net.mindview.util.Print.*;

// Definicja typu wyliczeniowego:
enum Signal { ZIELONE, ŻÓŁTE, CZERWONE, }

public class TrafficLight {
  Signal color = Signal.CZERWONE;
  public void change() {
    switch(color) {
      // Zauważ, że w klauzulach case nie trzeba
      // stosować zapisu Signal.CZERWONE itd.
      case CZERWONE:  color = Signal.ZIELONE;
                      break;
      case ZIELONE:   color = Signal.ŻÓŁTE;
                      break;
      case ŻÓŁTE:     color = Signal.CZERWONE;
                      break;
    }
```

```
  }
  public String toString() {
    return "Sygnalizator nadaje światło " + color;
  }
  public static void main(String[] args) {
    TrafficLight t = new TrafficLight();
    for(int i = 0; i < 7; i++) {
      print(t);
      t.change();
    }
  }
} /* Output:
Sygnalizator nadaje światło CZERWONE
Sygnalizator nadaje światło ZIELONE
Sygnalizator nadaje światło ŻÓŁTE
Sygnalizator nadaje światło CZERWONE
Sygnalizator nadaje światło ZIELONE
Sygnalizator nadaje światło ŻÓŁTE
Sygnalizator nadaje światło CZERWONE
*///:~
```

Kompilator nie protestuje wobec braku klauzuli `default` w instrukcji `switch`, ale wcale nie dlatego, że rozpoznał obecność klauzul `case` dla wszystkich nazw typu wyliczeniowego. Oznaczenie jako komentarz dowolnej z tych klauzul wciąż nie spowoduje błędu kompilacji. Trzeba więc samemu dbać o pokrycie wszystkich wartości typu wyliczeniowego w instrukcji wyboru. Z drugiej strony, gdyby instrukcje powrotu z metody znajdowały się w klauzulach `case`, kompilator oprotestowałby brak klauzuli domyślnej — nawet gdyby klauzule `case` pokrywały wszystkie możliwe wartości typu wyliczeniowego.

Ćwiczenie 1. Użyj importu statycznego do zmodyfikowania programu *TrafficLight.java*, tak aby nie trzeba było kwalifikować nazw typu wyliczeniowego (2).

Tajemnica metody values()

Rzekło się już, że wszystkie klasy typów wyliczeniowych są tworzone automatycznie przez kompilator jako pochodne klasy `Enum`. Jednakże spoglądając na klasę `Enum`, przekonasz się, że nie ma tam metody `values()`, a przecież używaliśmy jej z powodzeniem. Czyżby były tu jakieś metody ukryte? Możemy się o tym przekonać, pisząc prosty program bazujący na refleksji:

```
//: enumerated/Reflection.java
// Analiza typów wyliczeniowych.
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

enum Explore { HERE, THERE }

public class Reflection {
  public static Set<String> analyze(Class<?> enumClass) {
    print("----- Analiza: " + enumClass + " -----");
```

```
    print("Interfejsy:");
    for(Type t : enumClass.getGenericInterfaces())
      print(t);
    print("Klasa bazowa: " + enumClass.getSuperclass());
    print("Metody: ");
    Set<String> methods = new TreeSet<String>();
    for(Method m : enumClass.getMethods())
      methods.add(m.getName());
    print(methods);
    return methods;
  }
  public static void main(String[] args) {
    Set<String> exploreMethods = analyze(Explore.class);
    Set<String> enumMethods = analyze(Enum.class);
    print("Explore.containsAll(Enum)? " +
      exploreMethods.containsAll(enumMethods));
    printnb("Explore.removeAll(Enum): ");
    exploreMethods.removeAll(enumMethods);
    print(exploreMethods);
    // Dekompilacja kodu typu wyliczeniowego:
    OSExecute.command("javap Explore");
  }
} /* Output:
----- Analiza: class Explore -----
Interfejsy:
Klasa bazowa: class java.lang.Enum
Metody:
[compareTo, equals, getClass, getDeclaringClass, hashCode, name, notify, notifyAll, ordinal, toString,
valueOf, values, wait]
----- Analiza class java.lang.Enum -----
Interfejsy:
java.lang.Comparable<E>
interface java.io.Serializable
Klasa bazowa: class java.lang.Object
Metody:
[compareTo, equals, getClass, getDeclaringClass, hashCode, name, notify, notifyAll, ordinal, toString,
valueOf, wait]
Explore.containsAll(Enum)? true
Explore.removeAll(Enum): [values]
Compiled from "Reflection.java"
final class Explore extends java.lang.Enum{
  public static final Explore HERE;
  public static final Explore THERE;
  public static final Explore[] values();
  public static Explore valueOf(java.lang.String);
  static {};
}
*///:~
```

Jak widać, metoda `values()` to metoda statyczna dodawana przez kompilator. W procesie tworzenia typu wyliczeniowego klasa `Explore` jest też uzupełniana metodą `valueOf()`. To nieco mylące, bo i klasa `Enum` zawiera metodę `valueOf()`, ale tamta ma dwa argumenty, a metoda dodana — tylko jeden. Ale ponieważ posługujemy się tu nazwami metod, a nie ich sygnaturami, po wywołaniu `Explore.removeAll(Enum)` pozostaje nam jedynie `[values]`.

Na wyjściu widać, że klasa `Explore` została przez kompilator określona jako finalna, co oznacza, że po typie wyliczeniowym nie można dziedziczyć. Klasa ma też blok inicjalizacji statycznej, który z kolei — jak się niebawem okaże — można przedefiniowywać.

Z racji mechanizmu opisanego w rozdziale „Typy ogólne" dekompilator nie posiada pełnej informacji o klasie `Enum`, więc jako klasę bazową dla `Explore` pokazuje `Enum`, zamiast właściwego typu `Enum<Explore>`.

Ponieważ metoda `values()` to metoda statyczna, wstawiana do definicji typu wyliczeniowego przez kompilator, w razie rzutowania typu wyliczeniowego na typ `Enum` metoda `values()` będzie niedostępna. Zauważ jednak, że w klasie `Class` istnieje metoda `getEnumConstants()`, więc mimo braku metody `values()` w interfejsie `Enum` możemy mimo wszystko odwołać się do nazw typu wyliczeniowego za pośrednictwem obiektu `Class`:

```
//: enumerated/UpcastEnum.java
// Po rzutowaniu typu wyliczeniowego w górę tracimy metody values()

enum Search { HITHER, YON }

public class UpcastEnum {
  public static void main(String[] args) {
    Search[] vals = Search.values();
    Enum e = Search.HITHER; // Rzutowanie w górę
    // e.values(); // Brak metody values() w klasie Enum
    for(Enum en : e.getClass().getEnumConstants())
      System.out.println(en);
  }
} /* Output:
HITHER
YON
*///:~
```

Ponieważ metoda `getEnumConstants()` to metoda klasy `Class`, można ją wywołać również dla klasy niedefiniującej wyliczenia:

```
//: enumerated/NonEnum.java

public class NonEnum {
  public static void main(String[] args) {
    Class<Integer> intClass = Integer.class;
    try {
      for(Object en : intClass.getEnumConstants())
        System.out.println(en);
    } catch(Exception e) {
      System.out.println(e);
    }
  }
} /* Output:
java.lang.NullPointerException
*///:~
```

Metoda zwróci jednak wartość `null`, co przy próbie wypisania jej wyniku spowoduje wyjątek odwołania do referencji pustej.

Implementuje, nie dziedziczy

Ustaliliśmy, że wszystkie typy wyliczeniowe rozszerzają klasę `java.lang.Enum`. Ponieważ Java nie obsługuje dziedziczenia wielobazowego, nie można utworzyć dziedziczącego typu wyliczeniowego:

```
enum NotPossible extends Pet { ...    // Nie zadziała
```

Można jednak tworzyć typy wyliczeniowe implementujące dowolną liczbę interfejsów:

```
//: enumerated/cartoons/EnumImplementation.java
// Typ wyliczeniowy może implementować interfejsy
package enumerated.cartoons;
import java.util.*;
import net.mindview.util.*;

enum CartoonCharacter
implements Generator<CartoonCharacter> {
  REKSIO, BOLEK, LOLEK, TOLA, WILK, ZAJĄC, BOB;
  private Random rand = new Random(47);
  public CartoonCharacter next() {
    return values()[rand.nextInt(values().length)];
  }
}

public class EnumImplementation {
  public static <T> void printNext(Generator<T> rg) {
    System.out.print(rg.next() + ", ");
  }
  public static void main(String[] args) {
    // Wybierz dowolną nazwę:
    CartoonCharacter cc = CartoonCharacter.BOB;
    for(int i = 0; i < 10; i++)
      printNext(cc);
  }
} /* Output:
BOB, LOLEK, BOB, BOLEK, ZAJĄC, LOLEK, REKSIO, ZAJĄC, ZAJĄC, REKSIO,
*///:~
```

Nieco to dziwaczne, bo aby wywołać metodę, musimy dysponować egzemplarzem nazwy typu wyliczeniowego, na rzecz którego będzie można zrealizować wywołanie. Ale wartość typu `CartoonCharacter` może być przekazywana wszędzie tam, gdzie oczekuje się implementacji `Generator`, jak w metodzie `printNext()`.

Ćwiczenie 2. Zamiast implementować interfejs uczynić metodę `next()` metodą statyczną. Jakie są zalety i wady takiego podejścia (2)?

Wybór losowy

W wielu przykładach w tym rozdziale wymagany jest losowy wybór pomiędzy różnymi wartościami typu wyliczeniowego — jak choćby w `CartoonCharacter.next()`. Zadanie to można uogólnić przy użyciu nowych mechanizmów języka i umieścić rozwiązanie w ogólnodostępnej bibliotece:

```
//: net/mindview/util/Enums.java
package net.mindview.util;
import java.util.*;

public class Enums {
  private static Random rand = new Random(47);
  public static <T extends Enum<T>> T random(Class<T> ec) {
    return random(ec.getEnumConstants());
  }
  public static <T> T random(T[] values) {
    return values[rand.nextInt(values.length)];
  }
} ///:~
```

Nieco dziwaczna składnia `<T extends Enum<T>>` opisuje typ `T` jako wartość typu wyliczeniowego. Przekazując `Class<T>`, udostępniamy w metodzie obiekt klasy, z którego można wydobyć tablicę wartości wyliczenia. Przeciążona metoda `random()` przyjmuje z kolej tablicę `T[]`, bo nie musi wykonywać żadnych operacji na typie `Enum` — ma jedynie wybrać losowy element tablicy. Typ wartości zwracanej jest zgodny z dokładnym typem wartości wyliczenia.

Oto prosty test działania metody `random()`:

```
//: enumerated/RandomTest.java
import net.mindview.util.*;

enum Activity { SITTING, LYING, STANDING, HOPPING,
  RUNNING, DODGING, JUMPING, FALLING, FLYING }

public class RandomTest {
  public static void main(String[] args) {
    for(int i = 0; i < 20; i++)
      System.out.print(Enums.random(Activity.class) + " ");
  }
} /* Output:
STANDING FLYING RUNNING STANDING RUNNING STANDING LYING DODGING SITTING
RUNNING HOPPING HOPPING HOPPING RUNNING STANDING LYING FALLING RUNNING
FLYING LYING
*///:~
```

Choć `Enum` to prosta klasa, przekonasz się, że dzięki niej unikniemy w przykładach znacznej ilości powtórzeń kodu. A wiadomo, że każde powtórzenie to powtórna szansa na błąd, więc gra jest warta świeczki.

Organizacja na bazie interfejsów

Niemożność dziedziczenia po typie wyliczeniowym może być niekiedy frustrująca. Powodem takiej chęci jest zwykle albo potrzeba rozszerzenia zakresu wartości pierwotnego typu wyliczeniowego, albo chęć tworzenia podkategorii tego zakresu.

Kategoryzację można zrealizować przez grupowanie elementów w obrębie interfejsów i tworzenie wyliczeń w oparciu o te interfejsy. Załóżmy na przykład, że dysponujemy różnymi klasami reprezentującymi posiłki (Food), które chcielibyśmy traktować jako typy wyliczeniowe, ale aby każde z takich wyliczeń było typu Food. Wyglądałoby to tak:

```
//: enumerated/menu/Food.java
// Podkategorie wyliczenia w obrębie interfejsów.
package enumerated.menu;

public interface Food {
  enum Appetizer implements Food {
    SALAD, SOUP, SPRING_ROLLS;
  }
  enum MainCourse implements Food {
    LASAGNE, BURRITO, PAD_THAI,
    LENTILS, HUMMOUS, VINDALOO;
  }
  enum Dessert implements Food {
    TIRAMISU, GELATO, BLACK_FOREST_CAKE,
    FRUIT, CREME_CARAMEL;
  }
  enum Coffee implements Food {
    BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
    LATTE, CAPPUCCINO, TEA, HERB_TEA;
  }
} ///:~
```

Ponieważ jedyną dostępną dla typów wyliczeniowych techniką kategoryzacji jest implementacja interfejsów, każdy zagnieżdżony typ wyliczeniowy implementuje nadrzędny interfejs Food. Teraz można powiedzieć, że „wszystko jest typu Food", jak tutaj:

```
//: enumerated/menu/TypeOfFood.java
package enumerated.menu;
import static enumerated.menu.Food.*;

public class TypeOfFood {
  public static void main(String[] args) {
    Food food = Appetizer.SALAD;
    food = MainCourse.LASAGNE;
    food = Dessert.GELATO;
    food = Coffee.CAPPUCCINO;
  }
} ///:~
```

Rzutowanie w górę na typ Food działa dla każdego typu wyliczeniowego implementującego interfejs Food, więc wszystkie te typy są podtypami Food.

Interfejs nie jest jednak tak przydatny jak typ wyliczeniowy, jeśli chodzi o zestawy typów. Gdybyśmy chcieli utworzyć „wyliczenie wyliczeń", moglibyśmy utworzyć zewnętrzny typ wyliczeniowy z jedną wartością dla każdego typu wyliczeniowego w obrębie Food:

```
//: enumerated/menu/Course.java
package enumerated.menu;
import net.mindview.util.*;

public enum Course {
  APPETIZER(Food.Appetizer.class),
  MAINCOURSE(Food.MainCourse.class),
  DESSERT(Food.Dessert.class),
  COFFEE(Food.Coffee.class);
  private Food[] values;
  private Course(Class<? extends Food> kind) {
    values = kind.getEnumConstants();
  }
  public Food randomSelection() {
    return Enums.random(values);
  }
} ///:~
```

Każdy z powyższych typów wyliczeniowych przyjmuje w wywołaniu konstruktora odpowiedni egzemplarz Class, z którego może wyłuskać i zachować wszystkie wartości wyliczenia — za pośrednictwem metody getEnumConstants(). Wartości te są potem wykorzystywane w metodzie randomSelection(), dzięki której możemy tworzyć losowo komponowane posiłki, wybierając po jednym elemencie Food z każdego typu Course (danie):

```
//: enumerated/menu/Meal.java
package enumerated.menu;

public class Meal {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      for(Course course : Course.values()) {
        Food food = course.randomSelection();
        System.out.println(food);
      }
      System.out.println("---");
    }
  }
} /* Output:
SPRING_ROLLS
VINDALOO
FRUIT
DECAF_COFFEE
---
SOUP
VINDALOO
FRUIT
TEA
---
SALAD
BURRITO
FRUIT
TEA
---
SALAD
```

```
BURRITO
CREME_CARAMEL
LATTE
---
SOUP
BURRITO
TIRAMISU
ESPRESSO
---
*///:~
```

W tym przypadku zaletą tworzenia „wyliczenia wyliczeń" jest możliwość przeglądania wszystkich egzemplarzy Course. Nieco później, w przykładzie *VendingMachine.java*, zobaczymy inne podejście do kategoryzacji, wedle innych ograniczeń.

Inne podejście do kategoryzacji polega na zagnieżdżaniu typów wyliczeniowych w innych typach wyliczeniowych, jak tu:

```
//: enumerated/SecurityCategory.java
// Zwięzłe wydzielanie podkategorii typów wyliczeniowych.
import net.mindview.util.*;

enum SecurityCategory {
  STOCK(Security.Stock.class), BOND(Security.Bond.class);
  Security[] values;
  SecurityCategory(Class<? extends Security> kind) {
    values = kind.getEnumConstants();
  }
  interface Security {
    enum Stock implements Security { SHORT, LONG, MARGIN }
    enum Bond implements Security { MUNICIPAL, JUNK }
  }
  public Security randomSelection() {
    return Enums.random(values);
  }
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++) {
      SecurityCategory category =
        Enums.random(SecurityCategory.class);
      System.out.println(category + ": " +
        category.randomSelection());
    }
  }
} /* Output:
BOND: MUNICIPAL
BOND: MUNICIPAL
STOCK: MARGIN
STOCK: MARGIN
BOND: JUNK
STOCK: SHORT
STOCK: LONG
STOCK: LONG
BOND: MUNICIPAL
BOND: JUNK
*///:~
```

Interfejs Security jest niezbędny do zebrania wszystkich typów wyliczeniowych w jednym wspólnym typie. Kategoryzacja odbywa się wtedy za pośrednictwem wyliczenia SecurityCategory.

Gdybyśmy tę samą technikę zastosowali w przykładzie z klasą Food, otrzymalibyśmy coś takiego:

```
//: enumerated/menu/Meal2.java
package enumerated.menu;
import net.mindview.util.*;

public enum Meal2 {
  APPETIZER(Food.Appetizer.class),
  MAINCOURSE(Food.MainCourse.class),
  DESSERT(Food.Dessert.class),
  COFFEE(Food.Coffee.class);
  private Food[] values;
  private Meal2(Class<? extends Food> kind) {
    values = kind.getEnumConstants();
  }
  public interface Food {
    enum Appetizer implements Food {
      SALAD, SOUP, SPRING_ROLLS;
    }
    enum MainCourse implements Food {
      LASAGNE, BURRITO, PAD_THAI,
      LENTILS, HUMMOUS, VINDALOO;
    }
    enum Dessert implements Food {
      TIRAMISU, GELATO, BLACK_FOREST_CAKE,
      FRUIT, CREME_CARAMEL;
    }
    enum Coffee implements Food {
      BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
      LATTE, CAPPUCCINO, TEA, HERB_TEA;
    }
  }
  public Food randomSelection() {
    return Enums.random(values);
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      for(Meal2 meal : Meal2.values()) {
        Food food = meal.randomSelection();
        System.out.println(food);
      }
      System.out.println("---");
    }
  }
} /* Zobacz Meal.java *///:~
```

Niby to tylko reorganizacja kodu, ale w niektórych przypadkach może zaowocować czytelniejszą strukturą.

Ćwiczenie 3. Dodaj do programu *Course.java* nowy element wyliczenia Course i pokaż, że działa on w programie *Meal.java* (1).

Ćwiczenie 4. Powtórz powyższe ćwiczenie dla programu *Meal2.java* (1).

Ćwiczenie 5. Zmodyfikuj program *control/VowelsAndConsonants.java* tak, aby wykorzystywał trzy typy wyliczeniowe: `VOWEL`, `SOMETIMES_A_VOWEL` i `CONSONANT`. Konstruktor typu wyliczeniowego powinien przyjmować różne litery opisujące tą konkretną kategorię. Wskazówka: skorzystaj ze zmiennych list argumentów i pamiętaj, że listy te są automatycznie przekazywane w postaci tablic (4).

Ćwiczenie 6. Czy zagnieżdżanie typów wyliczeniowych `Appetizer`, `MainCourse`, `Dessert` i `Coffee` w interfejsie `Food` ma jakieś zalety wobec zwykłego implementowania interfejsu `Food` w samodzielnych typach wyliczeniowych (3)?

EnumSet zamiast znaczników

Klasa `Set` implementuje zbiór — rodzaj kolekcji, w ramach której mogą być przechowywane tylko unikaty danego typu. Typ wyliczeniowy również wymaga, aby jego wartości nie powtarzały się, zdaje się więc przejawiać zachowanie typowe dla zbioru, ale nie jest użyteczny, bo nie można ani dodawać do niego elementów, ani ich z niego usuwać. W Javie SE5 znalazła się jednak klasa `EnumSet`, bazująca na typach wyliczeniowych i przewidziana jako następca „pól bitowych" czy też „zestawów znaczników", implementowanych tradycyjnie na wartościach typu `int`. Znaczniki są typowo wykorzystywane jako przełączniki dwustanowe, ale w tradycyjnej implementacji wszystko kończy się na manipulowaniu anonimowymi bitami, co zaciemnia kod.

Zbiór `EnumSet` został zaprojektowany pod kątem szybkości, bo przecież musi jakoś konkurować ze znacznikami implementowanymi na bitach (operacje na zbiorze `EnumSet` są typowo znacznie szybsze niż analogiczne operacje na zbiorze `HashSet`). Wewnętrznie zbiór znaczników jest reprezentowany przez pojedynczą (o ile to możliwe) wartość typu `long`, traktowaną jako wektor bitowy; daje to efektywność i niezrównaną szybkość. Zaletą stosowania takiego zbioru jest możliwość jasnego wyrażania w kodzie obecności bądź braku cech binarnych.

Elementy zbioru `EnumSet` muszą wywodzić się ze wspólnego typu wyliczeniowego. Poniższy przykład wykorzystuje typ wyliczeniowy, którego wartości reprezentują miejsca montażu czujników alarmowych w budynku:

```
//: enumerated/AlarmPoints.java
package enumerated;
public enum AlarmPoints {
  STAIR1, STAIR2, LOBBY, OFFICE1, OFFICE2, OFFICE3,
  OFFICE4, BATHROOM, UTILITY, KITCHEN
} ///:~
```

Zbiór `EnumSet` może być wtedy wykorzystany do reprezentacji stanu alarmu:

```
//: enumerated/EnumSets.java
// Operacje na zbiorze EnumSets
package enumerated;
import java.util.*;
import static enumerated.AlarmPoints.*;
import static net.mindview.util.Print.*;
```

```
public class EnumSets {
  public static void main(String[] args) {
    EnumSet<AlarmPoints> points =
      EnumSet.noneOf(AlarmPoints.class); // Zbiór pusty
    points.add(BATHROOM);
    print(points);
    points.addAll(EnumSet.of(STAIR1, STAIR2, KITCHEN));
    print(points);
    points = EnumSet.allOf(AlarmPoints.class);
    points.removeAll(EnumSet.of(STAIR1, STAIR2, KITCHEN));
    print(points);
    points.removeAll(EnumSet.range(OFFICE1, OFFICE4));
    print(points);
    points = EnumSet.complementOf(points);
    print(points);
  }
} /* Output:
[BATHROOM]
[STAIR1, STAIR2, BATHROOM, KITCHEN]
[LOBBY, OFFICE1, OFFICE2, OFFICE3, OFFICE4, BATHROOM, UTILITY]
[LOBBY, BATHROOM, UTILITY]
[STAIR1, STAIR2, OFFICE1, OFFICE2, OFFICE3, OFFICE4, KITCHEN]
*///:~
```

Statyczny import służy tu do uproszczenia stosowania stałych typu wyliczeniowego. Nazwy metod mówią same za siebie, a szczegółów można poszukać w dokumentacji JDK. Kiedy do niej zajrzysz, zauważysz zapewne coś ciekawego — metodę `of()` przeciążoną dla zmiennej listy argumentów i dla różnych zestawów argumentów, od dwóch do pięciu sztuk. Już to przeciążenie sugeruje troskę o wydajność obsługi zbioru `EnumSet`, bo pojedyncza metoda `of()` dla zmiennej liczby argumentów wystarczyłaby w zupełności, ale przekazywanie zmiennej listy argumentów jest nieco mniej efektywne niż wywołanie z jawną listą argumentów. Jeśli argumentów wywołania będzie więcej niż pięć, wybrana zostanie i tak wersja przeciążona dla zmiennej listy argumentów. Zauważ, że w wywołaniu z pojedynczym argumentem kompilator nie skonstruuje tablicy argumentów, więc unikniemy narzutu typowego dla metod ze zmiennymi listami argumentów.

Zbiory `EnumSet` bazują na wartościach podstawowego typu `long`; typ `long` ma rozmiar 64 bitów, a każda wartość wyliczenia wymaga reprezentacji obecności bądź braku za pomocą pojedynczego bitu. Oznacza to, że zbiór `EnumSet` przeznaczony do obsługi więcej niż 64 znaczników nie zmieści się na jednej wartości `long`. Co się wtedy stanie?

```
//: enumerated/BigEnumSet.java
import java.util.*;

public class BigEnumSet {
  enum Big { A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10,
    A11, A12, A13, A14, A15, A16, A17, A18, A19, A20, A21,
    A22, A23, A24, A25, A26, A27, A28, A29, A30, A31, A32,
    A33, A34, A35, A36, A37, A38, A39, A40, A41, A42, A43,
    A44, A45, A46, A47, A48, A49, A50, A51, A52, A53, A54,
    A55, A56, A57, A58, A59, A60, A61, A62, A63, A64, A65,
    A66, A67, A68, A69, A70, A71, A72, A73, A74, A75 }
  public static void main(String[] args) {
    EnumSet<Big> bigEnumSet = EnumSet.allOf(Big.class);
    System.out.println(bigEnumSet);
  }
```

```
} /* Output:
[A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15, A16, A17, A18, A19, A20, A21,
A22, A23, A24, A25, A26, A27, A28, A29, A30, A31, A32, A33, A34, A35, A36, A37, A38, A39, A40, A41,
A42, A43, A44, A45, A46, A47, A48, A49, A50, A51, A52, A53, A54, A55, A56, A57, A58, A59, A60, A61,
A62, A63, A64, A65, A66, A67, A68, A69, A70, A71, A72, A73, A74, A75]
*///:~
```

Najwyraźniej klasa `EnumSet` świetnie radzi sobie z typami wyliczeniowymi o więcej niż 64 elementach, możemy więc założyć, że w miarę potrzeby dokłada sobie kolejne wartości `long`.

Ćwiczenie 7. Odszukaj kod źródłowy klasy `EnumSet` i wyjaśnij, jak działa ta klasa (zwłaszcza odnośnie większej liczby znaczników) (3).

Stosowanie klasy EnumMap

Klasa `EnumMap` to specjalizowany kontener asocjacyjny, wymagający, aby klucze przechowywanych par były wartościami pojedynczego typu wyliczeniowego. Z racji ograniczeń typów wyliczeniowych `EnumMap` może być wewnętrznie implementowany na bazie tablicy. Dzięki temu jest kontenerem wyjątkowo szybkim i znakomicie nadającym się do kojarzenia wartości typów wyliczeniowych z obiektami.

Metodę `put()` można wywoływać jedynie dla kluczy z obrębu typu wyliczeniowego — poza tym kontener stosuje się tak jak inne kontenery `Map`.

Oto przykład ilustrujący zastosowanie wzorca projektowego *Command* („polecenie"). Wzorzec ten zakłada obecność interfejsu zawierającego (zazwyczaj) pojedynczą metodę i tworzy wiele implementacji tej metody różniących się zachowaniem. Wdrożenie wzorca polega na zainstalowaniu obiektów-poleceń, aby program mógł wywoływać je wedle potrzeb:

```
//: enumerated/EnumMaps.java
// Podstawy stosowania klasy EnumMaps.
package enumerated;
import java.util.*;
import static enumerated.AlarmPoints.*;
import static net.mindview.util.Print.*;

interface Command { void action(); }

public class EnumMaps {
  public static void main(String[] args) {
    EnumMap<AlarmPoints,Command> em =
      new EnumMap<AlarmPoints,Command>(AlarmPoints.class);
    em.put(KITCHEN, new Command() {
      public void action() { print("Pożar kuchni!"); }
    });
    em.put(BATHROOM, new Command() {
      public void action() { print("Alarm w łazience!"); }
    });
    for(Map.Entry<AlarmPoints,Command> e : em.entrySet()) {
      printnb(e.getKey() + ": ");
      e.getValue().action();
```

```
    }
    try { // W przypadku braku wartości dla danego klucza:
      em.get(UTILITY).action();
    } catch(Exception e) {
      print(e);
    }
  }
} /* Output:
BATHROOM: Alarm w łazience!
KITCHEN: Pożar kuchni!
java.lang.NullPointerException
*///:~
```

Tak jak w przypadku `EnumSet` kolejność elementów w `EnumMap` jest określona przez kolejność definicji wartości typu wyliczeniowego.

Ostatnia część metody `main()` pokazuje, że każda wartość typu wyliczeniowego posiada swój element w kontenerze, choć dopóki nie zostanie dla niej wywołana metoda `put()`, odwołanie do elementu zwraca wartość pustą (`null`).

W implementacji wzorca projektowego *Command* kontener `EnumMap` ma przewagę wobec specjalizacji *metod dla poszczególnych stałych wyliczenia* (zobacz następny podrozdział), bo pozwala na zmienianie obiektów wartości, podczas gdy metody specjalizowane dla wartości wyliczenia są ustalane w czasie kompilacji.

Przekonasz się, że kontenery `EnumMap` można wykorzystać do realizacji *wielokrotnego rozprowadzania* (ang. *multiple dispatching*) w sytuacjach, gdzie występuje wiele typów wyliczeniowych wchodzących z sobą w interakcje.

Metody specjalizowane dla elementów wyliczenia

Typy wyliczeniowe w języku Java przejawiają bardzo ciekawą cechę w postaci możliwości różnicowania zachowania poszczególnych wartości typu wyliczeniowego przez tworzenie metod specjalizowanych dla tych wartości. Aby to osiągnąć, należy w typie wyliczeniowym zdefiniować jedną lub kilka metod abstrakcyjnych, a następnie zdefiniować te metody dla poszczególnych wartości wyliczenia. Oto przykład:

```
//: enumerated/ConstantSpecificMethod.java
import java.util.*;
import java.text.*;

public enum ConstantSpecificMethod {
  DATE_TIME {
    String getInfo() {
      return
        DateFormat.getDateInstance().format(new Date());
    }
  },
  CLASSPATH {
```

```
    String getInfo() {
      return System.getenv("CLASSPATH");
    }
  },
  VERSION {
    String getInfo() {
      return System.getProperty("java.version");
    }
  };
  abstract String getInfo();
  public static void main(String[] args) {
    for(ConstantSpecificMethod csm : values())
      System.out.println(csm.getInfo());
  }
} /* (Execute to see output) *///:~
```

Można tu wybierać metody do wywołania na podstawie wartości typu wyliczeniowego. Takie programowanie nosi często miano *kodowania tabelowego* (zwróć uwagę na podobieństwo do wspomnianego niedawno wzorca projektowego *Command*).

W programowaniu obiektowym różnicowanie zachowania wynika ze zróżnicowania klas obiektów. Skoro każda wartość typu wyliczeniowego może definiować własne zachowanie na bazie specjalizowanych dla siebie metod, znaczyłoby to, że każda z tych wartości stanowi osobny typ. W powyższym przykładzie każda wartość typu wyliczeniowego jest traktowana jako wartość „typu bazowego" `ConstantSpecificMethod`, ale wywołanie metody `getInfo()` dla poszczególnych wartości daje zachowanie typowe dla polimorfizmu.

Ale na tym kończą się podobieństwa; wartości typu wyliczeniowego nie można traktować jako osobnych klas:

```
//: enumerated/NotClasses.java
// {Exec: javap -c LikeClasses}
import static net.mindview.util.Print.*;

enum LikeClasses {
  WINKEN { void behavior() { print("Zachowanie1"); } },
  BLINKEN { void behavior() { print("Zachowanie2"); } },
  NOD { void behavior() { print("Zachowanie3"); } };
  abstract void behavior();
}

public class NotClasses {
  // void f1(LikeClasses.WINKEN instance) {} // Nic z tego
} /* Output:
Compiled from "NotClasses.java"
abstract class LikeClasses extends java.lang.Enum{
public static final LikeClasses WINKEN;

public static final LikeClasses BLINKEN;

public static final LikeClasses NOD;
...
*///:~
```

W metodzie f1() kompilator nie pozwala na użycie wartości typu wyliczeniowego jako typu klasowego, co staje się oczywiste po analizie kodu wygenerowanego przez kompilator dla tego typu wyliczeniowego — poszczególne wartości typu wyliczeniowego są tu statycznymi, finalnymi egzemplarzami LikeClasses.

Wartości wewnętrznego typu wyliczeniowego, skoro są statyczne, nie mogą zachowywać się jak zwykłe klasy wewnętrzne — nie można odwołać się w nich do niestatycznych pól i metod klasy otaczającej.

W ramach jeszcze ciekawszego przykładu rozważmy model myjni samochodowej. Każdy klient otrzymuje możliwość wyboru trybów mycia, a każdy tryb przewiduje wykonanie różnych czynności. Z każdym trybem skojarzona jest metoda specjalizowana dla wartości typu wyliczeniowego reprezentującego ten tryb; do przechowywania wyborów klientów można wykorzystać kontener EnumSet:

```
// enumerated/CarWash.java
import java.util.*;
import static net.mindview.util.Print.*;

public class CarWash {
  public enum Cycle {
    UNDERBODY {
      void action() { print("Ciśnieniowe płukanie podwozia"); }
    },
    WHEELWASH {
      void action() { print("Mycie kół"); }
    },
    PREWASH {
      void action() { print("Płukanie wstępne"); }
    },
    BASIC {
      void action() { print("Mycie zasadnicze"); }
    },
    HOTWAX {
      void action() { print("Woskowanie na gorąco"); }
    },
    RINSE {
      void action() { print("Płukanie"); }
    },
    BLOWDRY {
      void action() { print("Suszenie"); }
    };
    abstract void action();
  }
  EnumSet<Cycle> cycles =
    EnumSet.of(Cycle.BASIC, Cycle.RINSE);
  public void add(Cycle cycle) { cycles.add(cycle); }
  public void washCar() {
    for(Cycle c : cycles)
      c.action();
  }
  public String toString() { return cycles.toString(); }
  public static void main(String[] args) {
    CarWash wash = new CarWash();
    print(wash);
    wash.washCar();
```

```
    // Kolejność dodawania do kontenera jest nieistotna:
    wash.add(Cycle.BLOWDRY);
    wash.add(Cycle.BLOWDRY); // Duplikaty są ignorowane
    wash.add(Cycle.RINSE);
    wash.add(Cycle.HOTWAX);
    print(wash);
    wash.washCar();
  }
} /* Output:
[BASIC, RINSE]
Mycie zasadnicze
Płukanie
[BASIC, HOTWAX, RINSE, BLOWDRY]
Mycie zasadnicze
Woskowanie na gorąco
Płukanie
Suszenie
*///:~
```

Składnia definiowania metod specjalizowanych dla wartości typu wyliczeniowego pokrywa się zasadniczo ze składnią klas wewnętrznych, ale jest jeszcze bardziej zwarta.

Przykład ten pokazuje przy okazji kolejne cechy zbioru `EnumSet`. Ponieważ jest to zbiór, będzie przechowywał tylko pojedyncze egzemplarze poszczególnych elementów, więc wielokrotne wywołania `add()` dla duplikatów wartości typu wyliczeniowego są najzwyczajniej ignorowane (jest to uzasadnione, bo element taki reprezentuje znacznik bitowy, a ten można przerzucić w stan „obecny" tylko raz — potem można go już tylko wyzerować). Widać też, że kolejność dodawania wartości wyliczenia do zbioru `EnumSet` jest zupełnie nieistotna — na wyjściu uzyskamy i tak kolejność zgodną z kolejnością deklaracji wartości w definicji typu wyliczeniowego.

Czy, zamiast implementować metodę abstrakcyjną, dałoby się przesłaniać metody specjalizowane dla wartości typu wyliczeniowego? Owszem, jak poniżej:

```
//: enumerated/OverrideConstantSpecific.java
import static net.mindview.util.Print.*;

public enum OverrideConstantSpecific {
  NUT, BOLT,
  WASHER {
    void f() { print("Metoda przeciążona"); }
  };
  void f() { print("Zachowanie domyślne"); }
  public static void main(String[] args) {
    for(OverrideConstantSpecific ocs : values()) {
      printnb(ocs + ": ");
      ocs.f();
    }
  }
} /* Output:
NUT: Zachowanie domyślne
BOLT: Zachowanie domyślne
WASHER: Metoda przeciążona
*///:~
```

Choć typy wyliczeniowe nie pozwalają na wszystko, zasadniczo można z nimi eksperymentować tak jak ze zwyczajnymi klasami.

Typy wyliczeniowe w łańcuchu odpowiedzialności

Wzorzec projektowy *Chain of Responsibility* („łańcuch odpowiedzialności") zakłada utworzenie szeregu różnych rozwiązań problemu i połączenie ich w łańcuch. Kiedy napłynie żądanie, jest ono przekazywane wzdłuż łańcucha do momentu, w którym znajdzie się dla niego rozwiązanie.

Prostą realizację łańcucha odpowiedzialności można zaimplementować właśnie za pomocą typu wyliczeniowego i metod specjalizowanych dla jego wartości. Przeanalizujmy model urzędu pocztowego, który stara się obsługiwać każdą przesyłkę w możliwie ogólny sposób, ale w przypadku przesyłek szczególnych kontynuuje próbę obsługi na inne sposoby dopóty, dopóki nie okaże się, że dostarczenie jest niemożliwe. Każda próba obsługi to realizacja wzorca projektowego *Strategy* („strategia"), a cała kolejka takich prób to właśnie łańcuch odpowiedzialności.

Zaczniemy od opisu przesyłki. Wszystkie interesujące nas cechy mogą być wyrażone na bazie typów wyliczeniowych. Ponieważ obiekty przesyłek (`Mail`) będą generowane losowo, to aby zmniejszyć prawdopodobieństwo obsługi poste restante (`GeneralDelivery`), należy zadbać o utworzenie większej liczby przesyłek adresowanych (`NO`), przez co definicje typów wyliczeniowych wydają się na pierwszy rzut oka dość dziwaczne.

W klasie `Mail` widnieje metoda `randomMail()` tworząca losowe sekwencje przesyłek testowych. Metoda `generator()` generuje obiekt-implementację interfejsu `Iterable`, który za pomocą metody `randomMail()` generuje pewną sekwencję przesyłek udostępnianych kolejno wywołaniami metody `next()` na rzecz iteratora sekwencji. Taka konstrukcja pozwala na proste stosowanie pętli foreach dla sekwencji zwracanej z wywołania `Mail.generator()`:

```
//: enumerated/PostOffice.java
// Modelowanie urzędu pocztowego.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class Mail {
  // Wartości NO obniżają prawdopodobieństwo przy wyborze losowym:
  enum GeneralDelivery {YES,NO1,NO2,NO3,NO4,NO5}
  enum Scannability {UNSCANNABLE,YES1,YES2,YES3,YES4}
  enum Readability {ILLEGIBLE,YES1,YES2,YES3,YES4}
  enum Address {INCORRECT,OK1,OK2,OK3,OK4,OK5,OK6}
  enum ReturnAddress {MISSING,OK1,OK2,OK3,OK4,OK5}
  GeneralDelivery generalDelivery;
  Scannability scannability;
  Readability readability;
  Address address;
  ReturnAddress returnAddress;
  static long counter = 0;
  long id = counter++;
  public String toString() { return "Przesyłka " + id; }
  public String details() {
    return toString() +
      ". Poste restante: " + generalDelivery +
      ". Zdatność do czytnika: " + scannability +
```

```
      ", Czytelność adresu: " + readability +
      ", Adres: " + address +
      ", Adres zwrotny: " + returnAddress;
  }
  // Generowanie przesyłek testowych:
  public static Mail randomMail() {
    Mail m = new Mail();
    m.generalDelivery= Enums.random(GeneralDelivery.class);
    m.scannability = Enums.random(Scannability.class);
    m.readability = Enums.random(Readability.class);
    m.address = Enums.random(Address.class);
    m.returnAddress = Enums.random(ReturnAddress.class);
    return m;
  }
  public static Iterable<Mail> generator(final int count) {
    return new Iterable<Mail>() {
      int n = count;
      public Iterator<Mail> iterator() {
        return new Iterator<Mail>() {
          public boolean hasNext() { return n-- > 0; }
          public Mail next() { return randomMail(); }
          public void remove() { // Bez implementacji
            throw new UnsupportedOperationException();
          }
        };
      }
    };
  }
}

public class PostOffice {
  enum MailHandler {
    GENERAL_DELIVERY {
      boolean handle(Mail m) {
        switch(m.generalDelivery) {
          case YES:
            print(m + " - poste restante");
            return true;
          default: return false;
        }
      }
    },
    MACHINE_SCAN {
      boolean handle(Mail m) {
        switch(m.scannability) {
          case UNSCANNABLE: return false;
          default:
            switch(m.address) {
              case INCORRECT: return false;
              default:
                print(m + " - obsługa przez automat");
                return true;
            }
        }
      }
    },
    VISUAL_INSPECTION {
      boolean handle(Mail m) {
```

```
        switch(m.readability) {
          case ILLEGIBLE: return false;
          default:
            switch(m.address) {
              case INCORRECT: return false;
              default:
                print(m + " - obsługa ręczna");
                return true;
            }
        }
      }
    },
    RETURN_TO_SENDER {
      boolean handle(Mail m) {
        switch(m.returnAddress) {
          case MISSING: return false;
          default:
            print(m + " - zwrot do nadawcy");
            return true;
        }
      }
    };
    abstract boolean handle(Mail m);
  }
  static void handle(Mail m) {
    for(MailHandler handler : MailHandler.values())
      if(handler.handle(m))
        return;
    print(m + " - przesyłka martwa");
  }
  public static void main(String[] args) {
    for(Mail mail : Mail.generator(10)) {
      print(mail.details());
      handle(mail);
      print("*****");
    }
  }
} /* Output:
```

Przesyłka 0, Poste restante: NO2, Zdatność do czytnika: UNSCANNABLE, Czytelność adresu: YES3, Adres: OK1, Adres zwrotny: OK1
Przesyłka 0 - obsługa ręczna
*****
Przesyłka 1, Poste restante: NO5, Zdatność do czytnika: YES3, Czytelność adresu: ILLEGIBLE, Adres: OK5, Adres zwrotny: OK1
Przesyłka 1 - obsługa przez automat
*****
Przesyłka 2, Poste restante: YES, Zdatność do czytnika: YES3, Czytelność adresu: YES1, Adres: OK1, Adres zwrotny: OK5
Przesyłka 2 - poste restante
*****
Przesyłka 3, Poste restante: NO4, Zdatność do czytnika: YES3, Czytelność adresu: YES1, Adres: INCORRECT, Adres zwrotny: OK4
Przesyłka 3 - zwrot do nadawcy
*****
Przesyłka 4, Poste restante: NO4, Zdatność do czytnika: UNSCANNABLE, Czytelność adresu: YES1, Adres: INCORRECT, Adres zwrotny: OK2
Przesyłka 4 - zwrot do nadawcy
*****

```
Przesyłka 5, Poste restante: NO3, Zdatność do czytnika: YES1, Czytelność adresu: ILLEGIBLE, Adres:
OK4, Adres zwrotny: OK2
Przesyłka 5 - obsługa przez automat
*****
Przesyłka 6, Poste restante: YES, Zdatność do czytnika: YES4, Czytelność adresu: ILLEGIBLE, Adres:
OK4, Adres zwrotny: OK4
Przesyłka 6 - poste restante
*****
Przesyłka 7, Poste restante: YES, Zdatność do czytnika: YES3, Czytelność adresu: YES4, Adres: OK2,
Adres zwrotny: MISSING
Przesyłka 7 - poste restante
*****
Przesyłka 8, Poste restante: NO3, Zdatność do czytnika: YES1, Czytelność adresu: YES3, Adres:
INCORRECT, Adres zwrotny: MISSING
Przesyłka 8 - przesyłka martwa
*****
Przesyłka 9, Poste restante: NO1, Zdatność do czytnika: UNSCANNABLE, Czytelność adresu: YES2,
Adres: OK1, Adres zwrotny: OK4
Przesyłka 9 - obsługa ręczna
*****
*///:~
```

Łańcuch odpowiedzialności jest tu wyrażany w postaci typu wyliczeniowego `MailHandler`, a przepływ przesyłki wzdłuż łańcucha odpowiedzialności, czyli kolejność aplikowania kolejnych strategii obsługi, wynika z kolejności definicji wartości typu wyliczeniowego. Dla każdej przesyłki wypróbowywane są kolejne strategie, aż któraś okaże się skuteczna albo przesyłka zostanie uznana za „martwą".

Ćwiczenie 8. Zmodyfikuj program *PostOffice.java* tak, aby realizował również zadanie przekazywania poczty pod alternatywny adres (6).

Ćwiczenie 9. Zmodyfikuj klasę `PostOffice` tak, aby korzystała z kontenera `EnumMap` (5).

Projekt[2]. Specjalizowane języki programowania, takie jak Prolog, wykorzystują w rozwiązaniach podobnych problemów *wnioskowanie wstecz* (ang. *backward chaining*). Wzorując się na programie *PostOffice.java*, przeanalizuj podobne języki i opracuj program, który pozwoli na łatwe dodawanie do systemu nowych „reguł" obsługi.

Typy wyliczeniowe a automaty stanów

Typy wyliczeniowe nadają się idealnie do implementowania *automatów stanów* (ang. *state machine*). Automat stanów przechodzi z jednego stanu do następnego, dysponując skończoną liczbą stanów. Przejścia pomiędzy stanami odbywają się zazwyczaj w zależności od danych pojawiających się na wejściu, ale można wyróżnić również *stany tymczasowe* („ulotne", ang. *transient state*); z takich automat przechodzi do innych stanów bez pobudzania z zewnątrz.

Każdy stan dopuszcza pewien zakres danych wejściowych, a różne wartości tych danych wymuszają przechodzenie do różnych stanów następnych. Ponieważ typy wyliczeniowe ograniczają liczbę możliwych wartości, świetnie nadają się do reprezentowania tak różnych stanów automatu, jak i dopuszczalnych pobudzeń.

[2] Proponowane projekty można wykorzystać na przykład jako warunek zaliczenia semestru. Rozwiązania dostępne dla zwykłych ćwiczeń nie zawierają oczywiście rozwiązań projektów.

Każdy stan może być też skojarzony z jakimiś danymi wyjściowymi.

Dobrym przykładem automatu stanów jest kawomat czy dowolny automat sprzedający przysmaki czy bilety. Zdefiniujmy dla niego typ wyliczeniowy ograniczający impulsy wejściowe:

```
//: enumerated/Input.java
package enumerated;
import java.util.*;

public enum Input {
  PIĄTAK(5), DYCHA(10), DWUDZIESTKA(20), POŁÓWKA(50), ZŁOTY(100),
  PASTA(200), CHIPSY(75), COLA(100), MYDŁO(50),
  ABORT_TRANSACTION {
    public int amount() { // Niedozwolone
      throw new RuntimeException("ABORT.amount()");
    }
  },
  STOP { // To musi być ostatnia wartość.
    public int amount() { // Niedozwolone
      throw new RuntimeException("SHUT_DOWN.amount()");
    }
  };
  int value; // W groszach
  Input(int value) { this.value = value; }
  Input() {}
  int amount() { return value; }; // W groszach
  static Random rand = new Random(47);
  public static Input randomSelection() {
    // Bez STOP:
    return values()[rand.nextInt(values().length - 1)];
  }
} ///:~
```

Zauważ, że z niektórymi „impulsami wejściowymi" skojarzone są wartości pieniężne, stąd obecność metody `amount()` w interfejsie typu wyliczeniowego. Ale dla dwóch elementów typu wyliczeniowego `Input` wywołanie `amount()` jest niewłaściwe, więc definicje tej metody dla tych elementów powodują wyjątki. Definiowanie metody w interfejsie, a potem zgłaszanie wyjątku w razie wywołania tej metody dla niektórych implementacji jest cokolwiek dziwaczne, ale to wynik ograniczeń typów wyliczeniowych.

Automat `VendingMachine` reaguje na sygnały wejściowe, kategoryzując je za pośrednictwem typu wyliczeniowego `Category`, tak aby mógł wybierać (instrukcją `switch`) pomiędzy kategoriami. Poniższy przykład pokazuje, jak zastosowanie typów wyliczeniowych upraszcza kod i ułatwia jego konserwację:

```
//: enumerated/VendingMachine.java
// {Args: VendingMachineInput.txt}
package enumerated;
import java.util.*;
import net.mindview.util.*;
import static enumerated.Input.*;
import static net.mindview.util.Print.*;

enum Category {
  MONEY(PIĄTAK, DYCHA, DWUDZIESTKA, POŁÓWKA, ZŁOTY),
```

```
  ITEM_SELECTION(PASTA, CHIPSY, COLA, MYDŁO),
  QUIT_TRANSACTION(ABORT_TRANSACTION),
  SHUT_DOWN(STOP);
  private Input[] values;
  Category(Input... types) { values = types; }
  private static EnumMap<Input,Category> categories =
    new EnumMap<Input,Category>(Input.class);
  static {
    for(Category c : Category.class.getEnumConstants())
      for(Input type : c.values)
        categories.put(type, c);
  }
  public static Category categorize(Input input) {
    return categories.get(input);
  }
}

public class VendingMachine {
  private static State state = State.RESTING;
  private static int amount = 0;
  private static Input selection = null;
  enum StateDuration { TRANSIENT } // Znacznik
  enum State {
    RESTING {
      void next(Input input) {
        switch(Category.categorize(input)) {
          case MONEY:
            amount += input.amount();
            state = ADDING_MONEY;
            break;
          case SHUT_DOWN:
            state = TERMINAL;
          default:
        }
      }
    },
    ADDING_MONEY {
      void next(Input input) {
        switch(Category.categorize(input)) {
          case MONEY:
            amount += input.amount();
            break;
          case ITEM_SELECTION:
            selection = input;
            if(amount < selection.amount())
              print("Za mała kwota na " + selection);
            else state = DISPENSING;
            break;
          case QUIT_TRANSACTION:
            state = GIVING_CHANGE;
            break;
          case SHUT_DOWN:
            state = TERMINAL;
          default:
        }
      }
    },
```

```
    DISPENSING(StateDuration.TRANSIENT) {
      void next() {
        print("Oto Twoje " + selection);
        amount -= selection.amount();
        state = GIVING_CHANGE;
      }
    },
    GIVING_CHANGE(StateDuration.TRANSIENT) {
      void next() {
        if(amount > 0) {
          print("Reszta: " + amount);
          amount = 0;
        }
        state = RESTING;
      }
    },
    TERMINAL { void output() { print("Zatrzymany"); } };
    private boolean isTransient = false;
    State() {}
    State(StateDuration trans) { isTransient = true; }
    void next(Input input) {
      throw new RuntimeException("Metodę next(Input input)" +
        " wywołuj tylko dla stanów nieulotnych");
    }
    void next() {
      throw new RuntimeException("Metodę next() wywołuj jedynie dla " +
        "stanów StateDuration.TRANSIENT");
    }
    void output() { print(amount); }
  }
  static void run(Generator<Input> gen) {
    while(state != State.TERMINAL) {
      state.next(gen.next());
      while(state.isTransient)
        state.next();
      state.output();
    }
  }
  public static void main(String[] args) {
    Generator<Input> gen = new RandomInputGenerator();
    if(args.length == 1)
      gen = new FileInputGenerator(args[0]);
    run(gen);
  }
}

// Prosty test:
class RandomInputGenerator implements Generator<Input> {
  public Input next() { return Input.randomSelection(); }
}

// Generowanie wejść na podstawie pliku ciągów oddzielanych średnikami:
class FileInputGenerator implements Generator<Input> {
  private Iterator<String> input;
  public FileInputGenerator(String fileName) {
    input = new TextFile(fileName, ";").iterator();
  }
```

```
  public Input next() {
    if(!input.hasNext())
      return null;
    return Enum.valueOf(Input.class, input.next().trim());
  }
} /* Output:
20
40
60
Za mała kwota na CHIPSY
60
160
260
Oto Twoje PASTA
Reszta: 60
0
20
30
Reszta: 30
0
20
30
Za mała kwota na COLA
30
50
60
65
Za mała kwota na COLA
65
Reszta: 65
0
Zatrzymany
*///:~
```

Ponieważ wybór pomiędzy egzemplarzami typu wyliczeniowego odbywa się najczęściej w instrukcji wyboru `switch` (zwróć uwagę na dodatkowy wysiłek ze strony języka podejmowany w celu ułatwienia korzystania z typów wyliczeniowych w instrukcji `switch`), powstaje pytanie: „Co chcemy wybierać w instrukcji wyboru?". Tutaj łatwo na to pytanie odpowiedzieć, bo w każdym stanie `State` trzeba wybrać pomiędzy kategoriami pobudzeń wejściowych: przyjęciem bilonu do automatu wrzutowego, wyborem produktu do wydania, przerwaniem transakcji i wyłączeniem automatu. Ale w obrębie tych kategorii operuje się różnymi wartościami pieniężnymi i różnymi rodzajami produktów. Typ wyliczeniowy `Category` grupuje poszczególne rodzaje egzemplarzy `Input`, tak aby metoda `categorize()` mogła wygenerować odpowiednią kategorię w obrębie instrukcji wyboru `switch`. Metoda ta używa kontenera `EnumMap` w celu efektywnego i bezpiecznego wyboru.

Analiza klasy `VendingMachine` ujawnia, że każdy stan jest inny i inaczej reaguje na impulsy wejściowe. Zwróć też uwagę na dwa stany przejściowe: w metodzie `run()` automat oczekuje na egzemplarz `Input` i nie przerywa przechodzenia pomiędzy kolejnymi stanami aż do osiągnięcia następnego stanu trwałego.

Automat `VendingMachine` można przetestować na dwa sposoby, przy użyciu dwóch różnych obiektów-generatorów. Generator `RandomInputGenerator` ogranicza się do generowania impulsów wejściowych — wszystkich z wyjątkiem impulsu wyłączenia (`SHUT_DOWN`).

Uruchamiając ten generator na dłuższy czas, przeprowadzamy coś w rodzaju testu upewniającego co do niemożności zabłądzenia automatu do niepoprawnego stanu. Z kolei `FileInputGenerator` odczytuje ciągi impulsów wejściowych z pliku zewnętrznego, zamienia je na wartości typu wyliczeniowego i tworzy obiekty `Input`. Oto plik tekstowy, który posłużył do wygenerowania prezentowanego wyżej przebiegu programu:

```
//:! enumerated/VendingMachineInput.txt
DWUDZIESTKA; DWUDZIESTKA; DWUDZIESTKA; CHIPSY;
ZŁOTY; ZŁOTY; PASTA;
DWUDZIESTKA; DYCHA; ABORT_TRANSACTION;
DWUDZIESTKA; DYCHA; COLA;
DWUDZIESTKA; DYCHA; PIĄTAK; COLA;
ABORT_TRANSACTION;
STOP;
```

Pewnym ograniczeniem tego projektu jest to, że pola klasy `VendingMachine` dostępne dla egzemplarzy typu wyliczeniowego `State` *muszą* być polami statycznymi, co oznacza ograniczenie do pojedynczego egzemplarza klasy `VendingMachine`. W przypadku implementacji docelowej (w prawdziwym automacie) może to być zresztą ograniczenie zupełnie nieuciążliwe, bo i tak każdy automat będzie obsługiwany przez osobną aplikację.

Ćwiczenie 10. Zmodyfikuj klasę `VendingMachine` (tylko ją) za pomocą kontenera `EnumMap` tak, aby program mógł zawierać kilka egzemplarzy `VendingMachine` (7).

Ćwiczenie 11. W prawdziwym automacie wypadałoby udostępnić możliwość łatwego dodawania i modyfikowania typów wydawanych produktów, więc ograniczenia związane z zastosowaniem typu wyliczeniowego w `Input` są niepraktyczne (pamiętaj, że typ wyliczeniowy zakłada ograniczoną liczbę wartości). Zmodyfikuj program *VendingMachine.java* tak, aby sprzedawane przez automat produkty były reprezentowane klasami, i zainicjalizuj listę (`ArrayList`) takich obiektów na podstawie pliku tekstowego (korzystaj z pomocy klasy `net.mindview.util.TextFile`) (7).

Projekt[3]**.** Zaprojektuj automat z przystosowaniem do internacjonalizacji — tak aby automat dawało się przystosowywać do działania w różnych krajach.

Rozprowadzanie wielokrotne

Kiedy mamy do czynienia z wieloma typami, które wchodzą ze sobą w interakcje, program może się mocno powikłać. Za przykład niech posłuży system przetwarzający i obliczający wartości wyrażeń matematycznych. Chcemy w nim wyrażać operacje takie jak `Liczba.plus(Liczba)`, `Liczba.razy(Liczba)` i tym podobne, gdzie `Liczba` to jakaś klasa bazowa rozmaitych obiektów liczbowych. Ale jak zapewnić prawidłowe działanie wyrażenia `a.plus(b)`, kiedy nie znamy dokładnych typów ani obiektu `a`, ani obiektu `b`?

Odpowiedź zaczyna się od czegoś, czym rzadko się zajmujemy — mianowicie od stwierdzenia, że w zakresie rozprowadzania wywołań pomiędzy typami Java obsługuje jedynie *dyspozycję pojedynczą*. To znaczy, że operacja na obiektach, z których kilka jest

[3] Proponowane projekty można wykorzystać na przykład jako warunek zaliczenia semestru. Rozwiązania dostępne dla zwykłych ćwiczeń nie zawierają oczywiście rozwiązań projektów.

obiektami nieznanego typu, powoduje wykorzystanie mechanizmu dynamicznego wiązania tylko dla jednego z tych typów. Nie rozwiązuje to opisywanego problemu, więc trzeba ręcznie zastąpić i samodzielnie uzyskać efekt dynamicznego wiązania wywołań dla wielu typów.

Rozwiązaniem jest *rozprowadzanie wielokrotne* (albo *dyspozycja wielokrotna*, z ang. *multiple dispatching*; w naszym przypadku będzie to w zasadzie dyspozycja podwójna, a więc zaledwie *double dispatching*). Otóż polimorfizm ujawnia się wyłącznie w wywołaniach metod, więc chcąc uzyskać podwójne rozprowadzanie wywołania, trzeba skorzystać z dwóch wywołań metod: pierwsze określi pierwszy nieznany typ, a drugie określi drugi nieznany typ. Przy rozprowadzaniu wielokrotnym trzeba dysponować wirtualnym wywołaniem dla każdego z typów — jeśli rzecz dotyczy dwóch różnych hierarchii typów, potrzebne jest wirtualne wywołanie w obu tych hierarchiach. Zasadniczo chodzi o to, aby pojedyncze wywołanie metody generowało więcej niż jedno wywołanie wirtualne i w efekcie obsługiwało więcej niż jeden typ. Aby to osiągnąć, trzeba uciec się do wielu metod: potrzebne jest osobne wywołanie metody dla każdego rozprowadzania. Metody widniejące w poniższym przykładzie (implementującym grę w „papier, kamień i nożyczki" (gra ta jest też znana pod nazwą *RoShamBo*) noszą nazwy `compete()` i `eval()` i obie są składowymi tego samego typu. Generują one jeden z trzech możliwych wyników rozgrywki[4]:

```
//: enumerated/Outcome.java
package enumerated;
public enum Outcome {
  WIN {
    public String toString() { return "WYGRANA"; }
  },
  LOSE {
    public String toString() { return "PRZEGRANA"; }
  },
  DRAW {
    public String toString() { return "REMIS"; }
  }
} ///:~
```

```
//: enumerated/RoShamBo1.java
// Demonstracja dyspozycji wielokrotnej.
package enumerated;
import java.util.*;
import static enumerated.Outcome.*;

interface Item {
  Outcome compete(Item it);
  Outcome eval(Paper p);
  Outcome eval(Scissors s);
  Outcome eval(Rock r);
}

class Paper implements Item {
  public Outcome compete(Item it) { return it.eval(this); }
  public Outcome eval(Paper p) { return DRAW; }
```

[4] Przykład ten od lat wykorzystywałem w publikacjach *www.MindView.net* tak w C++, jak i w Javie (zobacz *Thinking in Patterns*), zanim jeszcze pojawił się w książkach innych autorów.

```
  public Outcome eval(Scissors s) { return WIN; }
  public Outcome eval(Rock r) { return LOSE; }
  public String toString() { return "Papier"; }
}

class Scissors implements Item {
  public Outcome compete(Item it) { return it.eval(this); }
  public Outcome eval(Paper p) { return LOSE; }
  public Outcome eval(Scissors s) { return DRAW; }
  public Outcome eval(Rock r) { return WIN; }
  public String toString() { return "Nożyczki"; }
}

class Rock implements Item {
  public Outcome compete(Item it) { return it.eval(this); }
  public Outcome eval(Paper p) { return WIN; }
  public Outcome eval(Scissors s) { return LOSE; }
  public Outcome eval(Rock r) { return DRAW; }
  public String toString() { return "Kamień"; }
}

public class RoShamBo1 {
  static final int SIZE = 20;
  private static Random rand = new Random(47);
  public static Item newItem() {
    switch(rand.nextInt(3)) {
      default:
      case 0: return new Scissors();
      case 1: return new Paper();
      case 2: return new Rock();
    }
  }
  public static void match(Item a, Item b) {
    System.out.println(
      a + " kontra " + b + ": " +  a.compete(b));
  }
  public static void main(String[] args) {
    for(int i = 0; i < SIZE; i++)
      match(newItem(), newItem());
  }
} /* Output:
Kamień kontra Kamień: REMIS
Papier kontra Kamień: WYGRANA
Papier kontra Kamień: WYGRANA
Papier kontra Kamień: WYGRANA
Nożyczki kontra Papier: WYGRANA
Nożyczki kontra Nożyczki: REMIS
Nożyczki kontra Papier: WYGRANA
Kamień kontra Papier: PRZEGRANA
Papier kontra Papier: REMIS
Kamień kontra Papier: PRZEGRANA
Papier kontra Nożyczki: PRZEGRANA
Papier kontra Nożyczki: PRZEGRANA
Kamień kontra Nożyczki: WYGRANA
Kamień kontra Papier: PRZEGRANA
papier kontra Kamień: WYGRANA
Nożyczki kontra Papier: WYGRANA
Papier kontra Nożyczki: PRZEGRANA
```

```
Papier kontra Nożyczki: PRZEGRANA
Papier kontra Nożyczki: PRZEGRANA
Papier kontra Nożyczki: PRZEGRANA
*///:~
```

Item to interfejs typów uczestniczących w rozprowadzaniu wielokrotnym. Metoda RoShamBo1.match() przyjmuje dwa obiekty Item i rozpoczyna proces podwójnego rozprowadzania, wywołując metodę Item.compete(). Mechanizm wywołania wirtualnego określa konkretny typ a, w wyniku czego wywołanie zostanie zrealizowane przez metodę compete() klasy właściwej dla a. Metoda compete() realizuje drugi etap rozprowadzania wywołania, wywołując metodę eval() na rzecz drugiego z typów. Argumentem tego wywołania jest sam obiekt wywołujący (this), co powoduje wybranie wersji przeciążonej odpowiedniej dla jego typu, a tym samym zachowanie informacji o typie z pierwszego rozprowadzenia. Po zakończeniu drugiego rozprowadzenia znamy już dokładne typy obu obiektów Item.

Ustawienie wielokrotnego rozprowadzania to cała ceremonia, ale włożony w nią wysiłek owocuje składniową elegancją widoczną w wywołaniu — miejsce dziwacznego kodu wykrywającego typ jednego czy wielu obiektów zajmuje żądanie: „Wy dwaj! Nie obchodzi mnie, jakiego typu jesteście, zgrajcie się z sobą!". Ale zanim przystąpisz do implementacji wielokrotnego rozprowadzania, zastanów się, czy na pewno potrzebujesz takiej elegancji.

Rozprowadzanie z udziałem typów wyliczeniowych

Proste przełożenie programu *RoShamBo1.java* na typy wyliczeniowe jest o tyle problematyczne, że wartości typów wyliczeniowych nie są same w sobie typami, więc nie można wedle nich przeciążać metody eval() — wartości typów wyliczeniowych nie mogą stanowić podstawy do różnicowania typów argumentów wywołania. Ale to nie znaczy, że nie da się zrealizować wielokrotnej dyspozycji z udziałem typów wyliczeniowych.

Jedna z metod wykorzystuje konstruktor inicjalizujący wartość typu wyliczeniowego „rzędem" wyników rozgrywki, co prowadzi do utworzenia rodzaju tabeli przeglądowej:

```
//: enumerated/RoShamBo2.java
// Przełączanie typu wyliczeniowego wedle innego typu wyliczeniowego.
package enumerated;
import static enumerated.Outcome.*;

public enum RoShamBo2 implements Competitor<RoShamBo2> {
  PAPER(DRAW, LOSE, WIN) {
    public String toString() { return "Papier"; }
  },
  SCISSORS(WIN, DRAW, LOSE) {
    public String toString() { return "Nożyczki"; }
  },
  ROCK(LOSE, WIN, DRAW) {
    public String toString() { return "Kamień"; }
  };
  private Outcome vPAPER, vSCISSORS, vROCK;
  RoShamBo2(Outcome paper,Outcome scissors,Outcome rock) {
    this.vPAPER = paper;
```

```
    this.vSCISSORS = scissors;
    this.vROCK = rock;
  }
  public Outcome compete(RoShamBo2 it) {
    switch(it) {
      default:
      case PAPER: return vPAPER;
      case SCISSORS: return vSCISSORS;
      case ROCK: return vROCK;
    }
  }
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo2.class, 20);
  }
} /* Output:
Kamień kontra Kamień: REMIS
Nożyczki kontra Kamień: PRZEGRANA
Nożyczki kontra Kamień: PRZEGRANA
Nożyczki kontra Kamień: PRZEGRANA
Papier kontra Nożyczki: PRZEGRANA
Papier kontra Papier: REMIS
Papier kontra Nożyczki: PRZEGRANA
Kamień kontra Nożyczki: WYGRANA
Nożyczki kontra Nożyczki: REMIS
Kamień kontra Nożyczki: WYGRANA
Nożyczki kontra Papier: WYGRANA
Nożyczki kontra Papier: WYGRANA
Kamień kontra Papier: PRZEGRANA
Kamień kontra Nożyczki: WYGRANA
Nożyczki kontra Kamień: PRZEGRANA
Papier kontra Nożyczki: PRZEGRANA
Nożyczki kontra Papier: WYGRANA
Nożyczki kontra Papier: WYGRANA
Nożyczki kontra Papier: WYGRANA
Nożyczki kontra Papier: WYGRANA
*///:~
```

Kiedy w metodzie `compete()` dookreślone zostają oba typy, pozostaje tylko zwrócić wynik rozgrywki `Outcome`. Ale można by też wywołać tam inną metodę, nawet (na przykład) na bazie obiektu „polecenia" (za wzorcem projektowym *Command*) ustalonego w konstruktorze.

Program *RoShamBo2.java* jest znacznie prostszy niż oryginał, a więc łatwiejszy do ogarnięcia. Zauważ jednak, że wciąż do określenia typów obu obiektów wykorzystujemy dwuetapowe rozprowadzanie. W *RoShamBo1.java* oba były realizowane na bazie wywołań wirtualnych, tu tylko pierwsze rozprowadzanie to wywołanie wirtualne. Drugie rozprowadzenie polega na wyborze w instrukcji `switch` — jest ono zupełnie bezpieczne, bo typ wyliczeniowy ogranicza wybory dostępne w ramach `switch`.

Kod związany z typem wyliczeniowym został wydzielony tak, aby mógł być ponownie wykorzystywany w kolejnych przykładach. Interfejs `Competitor` definiuje tu typ zdolny do konkurowania z innym obiektem `Competitor`:

```
//: enumerated/Competitor.java
// Przełączanie z typem wyliczeniowym.
package enumerated;
```

```
public interface Competitor<T extends Competitor<T>> {
  Outcome compete(T competitor);
} ///:~
```

Następnie definiujemy dwie metody statyczne (ich statyczność ma umożliwiać unikanie jawnego podawania parametru typowego). Pierwsza z nich, `match()`, wywołuje `compete()` na rzecz jednego obiektu `Competitor` z argumentem w postaci drugiego obiektu `Competitor`; parametr typowy to tu jedynie `Competitor<T>`. Ale już w metodzie `play()` parametr typowy musi być zarówno `Competitor<T>` (z racji przekazywania go do wywołania `match()`), jak i `Enum<T>` (z uwagi na użycie `Enums.random()`):

```
//: enumerated/RoShamBo.java
// Wspólne narzędzie rozgrywek RoShamBo.
package enumerated;
import net.mindview.util.*;

public class RoShamBo {
  public static <T extends Competitor<T>>
  void match(T a, T b) {
    System.out.println(
      a + " kontra " + b + ": " +  a.compete(b));
  }
  public static <T extends Enum<T> & Competitor<T>>
  void play(Class<T> rsbClass, int size) {
    for(int i = 0; i < size; i++)
      match(
        Enums.random(rsbClass),Enums.random(rsbClass));
  }
} ///:~
```

Metoda `play()` nie posiada wartości zwracanej angażującej parametr typowy T, co pozwala sądzić, że w typie `Class<T>` można by użyć symboli wieloznacznych. Symbole wieloznaczne jednak nie mogą rozszerzać więcej niż jednego typu bazowego, stąd konieczność zastosowanego wyrażenia.

Stosowanie metod specjalizowanych dla elementów wyliczenia

Skoro metody specjalizowane dla wartości typu wyliczeniowego pozwalają na różnicowanie implementacji metod tegoż typu dla poszczególnych jego elementów, to dlaczego nie użyć ich do realizacji rozprowadzania wielokrotnego? Niestety, choć w ten sposób faktycznie można różnicować zachowanie, to egzemplarze typu wyliczeniowego nie są typami i nie można używać ich jako typów w sygnaturach metod. Najlepsze, co można osiągnąć w naszym przykładzie, to ustawienie instrukcji wyboru:

```
//: enumerated/RoShamBo3.java
// Metody specjalizowane dla stałych wyliczenia.
package enumerated;
import static enumerated.Outcome.*;

public enum RoShamBo3 implements Competitor<RoShamBo3> {
  PAPER {
    public String toString() { return "Papier"; }
    public Outcome compete(RoShamBo3 it) {
```

```
      switch(it) {
        default: // Dla kompilatora
        case PAPER: return DRAW;
        case SCISSORS: return LOSE;
        case ROCK: return WIN;
      }
    }
  },
  SCISSORS {
    public String toString() { return "Nożyczki"; }
    public Outcome compete(RoShamBo3 it) {
      switch(it) {
        default:
        case PAPER: return WIN;
        case SCISSORS: return DRAW;
        case ROCK: return LOSE;
      }
    }
  },
  ROCK {
    public String toString() { return "Kamień"; }
    public Outcome compete(RoShamBo3 it) {
      switch(it) {
        default:
        case PAPER: return LOSE;
        case SCISSORS: return WIN;
        case ROCK: return DRAW;
      }
    }
  };
  public abstract Outcome compete(RoShamBo3 it);
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo3.class, 20);
  }
} /* Zobacz RoShamBo2.java *///:~
```

Choć zaprezentowane rozwiązanie działa i nie jest wcale najgorsze, wersja *RoShamBo2.java* wydaje się prostsza, choćby dlatego, że wymaga mniejszej ilości kodu przy dodawaniu nowego typu.

Jednakże przykład *RoShamBo3.java* można cokolwiek uprościć:

```
//: enumerated/RoShamBo4.java
package enumerated;

public enum RoShamBo4 implements Competitor<RoShamBo4> {
  ROCK {
    public String toString() { return "Kamień"; }
    public Outcome compete(RoShamBo4 opponent) {
      return compete(SCISSORS, opponent);
    }
  },
  SCISSORS {
    public String toString() { return "Nożyczki"; }
    public Outcome compete(RoShamBo4 opponent) {
      return compete(PAPER, opponent);
    }
```

```
  },
  PAPER {
    public String toString() { return "Papier"; }
    public Outcome compete(RoShamBo4 opponent) {
      return compete(ROCK, opponent);
    }
  };
  Outcome compete(RoShamBo4 loser, RoShamBo4 opponent) {
    return ((opponent == this) ? Outcome.DRAW
        : ((opponent == loser) ? Outcome.WIN
                               : Outcome.LOSE));
  }
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo4.class, 20);
  }
} /* Zobacz RoShamBo2.java *///:~
```

Drugie rozprowadzanie jest tu wykonywane na bazie dwuargumentowej wersji metody `compete()`, realizującej ciąg porównań i przez to podobnej w działaniu do instrukcji wyboru `switch`. Kod jest zwięźlejszy, ale mniej czytelny. W przypadku rozleglejszych systemów zmniejszenie czytelności może być zbyt uciążliwe.

Rozprowadzanie za pomocą EnumMap

Za pomocą klasy `EnumMap`, przystosowanej specjalnie do efektywnej obsługi typów wyliczeniowych, można osiągnąć „prawdziwe" rozprowadzanie dwukrotne. Skoro naszym celem jest przełączanie się pomiędzy dwoma nieznanymi typami, możemy zrealizować podwójne rozprowadzanie za pomocą kontenera `EnumMap` elementów typu `EnumMap`:

```
//: enumerated/RoShamBo5.java
// Rozprowadzanie wielokrotne odwzorowania odwzorowań EnumMap.
package enumerated;
import java.util.*;
import static enumerated.Outcome.*;

enum RoShamBo5 implements Competitor<RoShamBo5> {
  PAPER {
    public String toString() { return "Papier"; }
  },
  SCISSORS {
    public String toString() { return "Nożyczki"; }
  },
  ROCK {
    public String toString() { return "Kamień"; }
  };
  static EnumMap<RoShamBo5,EnumMap<RoShamBo5,Outcome>>
    table = new EnumMap<RoShamBo5,
      EnumMap<RoShamBo5,Outcome>>(RoShamBo5.class);
  static {
    for(RoShamBo5 it : RoShamBo5.values())
      table.put(it,
        new EnumMap<RoShamBo5,Outcome>(RoShamBo5.class));
    initRow(PAPER, DRAW, LOSE, WIN);
    initRow(SCISSORS, WIN, DRAW, LOSE);
    initRow(ROCK, LOSE, WIN, DRAW);
  }
```

```
  static void initRow(RoShamBo5 it,
    Outcome vPAPER, Outcome vSCISSORS, Outcome vROCK) {
    EnumMap<RoShamBo5,Outcome> row =
      RoShamBo5.table.get(it);
    row.put(RoShamBo5.PAPER, vPAPER);
    row.put(RoShamBo5.SCISSORS, vSCISSORS);
    row.put(RoShamBo5.ROCK, vROCK);
  }
  public Outcome compete(RoShamBo5 it) {
    return table.get(this).get(it);
  }
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo5.class, 20);
  }
} /* Zobacz RoShamBo2.java *///:~
```

Kontener `EnumMap` jest tu inicjalizowany w bloku inicjalizacji statycznej; widać „tabelową" strukturę wywołań `initRow()`. Zwróć uwagę na metodę `compete()`, w której oba etapy rozprowadzania złączyły się w pojedynczej instrukcji.

Z tablicą dwuwymiarową

Rozwiązanie możemy jeszcze bardziej uprościć, wykorzystując obserwację, że każdy z egzemplarzy typu wyliczeniowego posiada stałą wartość (zgodną z kolejnością występowania w deklaracji) i że wartość tę można uzyskać wywołaniem `ordinal()`. Najmniejszym i najprostszym rozwiązaniem problemu podwójnego rozprowadzania (i pewnie najszybszym, choć `EnumMap` wykorzystuje wewnętrznie tablicę) jest dwuwymiarowa tablica zawierająca wyniki rozgrywek poszczególnych „zawodników":

```
//: enumerated/RoShamBo6.java
// Wyliczenia z "tabelami" zamiast wielokrotnego rozprowadzania
package enumerated;
import static enumerated.Outcome.*;

enum RoShamBo6 implements Competitor<RoShamBo6> {
  PAPER {
    public String toString() { return "Papier"; }
  },
  SCISSORS {
    public String toString() { return "Nożyczki"; }
  },
  ROCK {
    public String toString() { return "Kamień"; }
  },
  private static Outcome[][] table = {
    { DRAW, LOSE, WIN }, // Papier
    { WIN, DRAW, LOSE }, // Nożyczki
    { LOSE, WIN, DRAW }, // Kamień
  };
  public Outcome compete(RoShamBo6 other) {
    return table[this.ordinal()][other.ordinal()];
  }
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo6.class, 20);
  }
} ///:~
```

Elementy tabeli `table` są ułożone w takiej samej kolejności jak w wywołaniach `initRow()` w poprzednim przykładzie.

Zwartość tego rozwiązania stanowi o jego atrakcyjności, zwłaszcza że w takiej postaci wydaje się znacznie łatwiejszy do ogarnięcia i ewentualnych modyfikacji. Ale nie jest aż tak różowo, głównie ze względu na brak „bezpieczeństwa" charakterystycznego dla poprzednich przykładów wynikający z użycia tablicy. Przy większej tablicy łatwo pomylić rozmiar, a jeśli testy aplikacji nie pokrywają wszystkich możliwych kombinacji, łatwo o błąd.

Wszystkie pokazane rozwiązania to różne rodzaje tabel, warto je jednak prześledzić, aby znaleźć najlepsze. Zauważ, że choć ostatnie pokazane rozwiązanie jest najbardziej zwarte, jest też mocno ograniczone w tym sensie, że dla stałych wejść daje stały wynik. Nic nie stoi jednak na przeszkodzie, aby tabela `table` zwracała obiekty funkcyjne. Tak czy inaczej, przekonałeś się chyba, że w pewnych kategoriach problemów sprawdzają się właśnie rozwiązania „tabelowe".

Podsumowanie

Choć typy wyliczeniowe jako takie nie są tworami wielce skomplikowanymi, niniejszy rozdział został celowo przesunięty na dość odległe miejsce w książce, bo możliwości manipulowania typami wyliczeniowymi wymagają opanowania polimorfizmu, typów ogólnych i refleksji.

Choć typy wyliczeniowe w Javie są zdecydowanie bardziej złożone niż ich odpowiedniki w C czy C++, wciąż należy je traktować jako niewielkie narzędzia pomocnicze, bez których język istniał (i odnosił sukcesy) przez wiele lat. Mimo to niniejszy rozdział pokazał chyba dobitnie, jaki wpływ na programowanie może mieć tak niewielkie udogodnienie — niekiedy samo w sobie wystarczające do skonstruowania eleganckiego i przejrzystego rozwiązania problemu. A znaczenie elegancji i czytelności rozwiązań podkreślałem już wielokrotnie — często to właśnie te czynniki dzielą rozwiązanie skuteczne i udane od nieudanego, bo niedającego się ogarnąć przez innych programistów.

Co do przejrzystości, to chciałbym od razu wyeliminować źródło pomieszania w postaci dość kiepskiego wyboru nazewnictwa w Javie 1.0, gdzie w miejsce „iteratora" — obiektu służącego do wybierania kolejnych elementów sekwencji (*kolekcji*) — występował „enumerator"; z kolei typy wyliczeniowe to z angielska „enumerations". W późniejszych wersjach Javy wycofano się z nieszczęśliwej terminologii, ale raz ustalonego interfejsu `Enumeration` (dla iteratorów) nie sposób przecież pozbyć się ot tak! Do dziś pokutuje on w starszych (a niekiedy i nowszych) programach, w bibliotece i dokumentacji.

Rozwiązania wybranych ćwiczeń można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 20.

Adnotacje

Adnotacje (zwane też metadanymi) to sformalizowany sposób uzupełniania kodu o informacje z przeznaczeniem do wykorzystania w przyszłości[1].

Adnotacje to wynik ogólniejszego trendu do osadzania w plikach kodu źródłowego metadanych wypierającego koncepcję przechowywania tych metadanych w zewnętrznych dokumentach. Adnotacje stanowią też reakcję na presję rozwojową ze strony języka C#.

Rzeczone adnotacje to jedne z tych zmian w Javie SE5, które mają charakter fundamentalny. Reprezentują one informacje potrzebne do pełnego opisu danego programu, których nie da się wyrazić wprost w kodzie źródłowym, a to ze względu na specyfikę języka Java. Adnotacje pozwalają więc na przechowywanie dodatkowych informacji o programie w formacie nadającym się do automatycznego testowania i weryfikacji ze strony kompilatora. Adnotacje można wykorzystywać do generowania plików opisu albo nawet definicji nowych klas, pomagają też w uniknięciu powtarzania „obowiązkowych", powtarzających się wciąż porcji kodu. Za pomocą adnotacji można wszystkie te dane osadzić w kodzie źródłowym i cieszyć się jego przejrzystością, kontrolą w czasie kompilacji oraz interfejsem programistycznym adnotacji, który wydatnie wspomaga konstruowanie narzędzi przetwarzających adnotacje. Choć w Javie SE5 niektóre typy metadanych zostały predefiniowane, to ogólnie rzecz biorąc, rodzaj dodawanych adnotacji i ich znaczenie zależą wyłącznie od programisty.

Składnia adnotacji jest dość prosta i ogranicza się w zasadzie do uzupełnienia języka o symbol `@`. Java SE5 definiuje trzy wbudowane adnotacje ogólnego przeznaczenia, zdefiniowane w pakiecie `java.lang`:

- `@Override` — adnotacja sygnalizująca, że definicja metody ma przesłaniać metodę z klasy bazowej. W takim układzie brak metody w klasie bazowej, wynikający choćby z błędnego zapisu nazwy czy sygnatury[2] przesłanianej metody, spowoduje błąd kompilacji.

---

[1] W pracach nad tym rozdziałem dołączył do mnie na całe dwa tygodnie Jeremy Meyer. Jego wkład jest bezcenny.

[2] To bez wątpienia wpływ obecności podobnego mechanizmu w języku C#. Otóż tam podobną rolę spełnia specjalne słowo kluczowe, a całość podlega ścisłej kontroli kompilatora. Kiedy w C# przesłaniamy metodę, słowo kluczowe `override` jest obowiązkowe; w Javie adnotacja `@Override` jest opcjonalna.

- `@Deprecated` — adnotacja powodująca ostrzeżenie ze strony kompilatora w przypadku użycia tak oznaczonego elementu.
- `@SuppressWarnings` — adnotacja tłumiąca niepożądane ostrzeżenia kompilatora. We wczesnych wydaniach Javy SE5 adnotacja ta jest dozwolona, ale nieobsługiwana (po prostu ignorowana).

Do tworzenia nowych adnotacji wydelegowano cztery inne adnotacje predefiniowane — poznasz je w dalszej części rozdziału.

Każdorazowo przy tworzeniu klas deskryptorów albo interfejsów, w których występują powtarzające się operacje, można skorzystać z adnotacji celem zautomatyzowania i uproszczenia powtórzeń. Większość takich powtórzeń w Enterprise JavaBeans (EJB) w wersji EJB3.0 jest eliminowana właśnie za pomocą adnotacji.

Adnotacje mogą zastąpić istniejące narzędzia w rodzaju XDoclet, czyli niezależne narzędzia dokumentujące (zobacz suplement publikowany pod adresem *http://www.MindView.net/Books/BetterJava*), specjalizowane właśnie pod kątem tworzenia „dokletów". Dla porównania adnotacje są elementami samego języka, dzięki czemu można korzystać z kontroli struktury i kontroli typów w czasie kompilacji. Przechowywanie wszystkich informacji wprost w kodzie źródłowym, a równocześnie poza komentarzami, zwiększa przejrzystość i ułatwia konserwację kodu. Korzystając z interfejsu programistycznego i specjalizowanych narzędzi obsługi adnotacji, a także rozszerzając je we własnym zakresie, zyskujemy możliwość podglądu i manipulowania nie tylko kodem źródłowym, ale i wynikowym kodem bajtowym.

Podstawy składni adnotacji

W poniższym przykładzie metoda `testExecute()` została opatrzona adnotacją `@Test`. Sama w sobie taka adnotacja jest zupełnie bezużyteczna, ale kompilator sprawdzi, czy w ścieżce kompilacji występuje definicja `@Test`. Jak się wkrótce okaże, pozwala to na tworzenie narzędzi uruchamiających daną metodę za pośrednictwem mechanizmu refleksji.

```
//: annotations/Testable.java
package annotations;
import net.mindview.atunit.*;

public class Testable {
  public void execute() {
    System.out.println("Wykonuję...");
  }
  @Test void testExecute() { execute(); }
} ///:~
```

Metody z adnotacjami nie różnią się niczym od zwyczajnych metod. Adnotacja `@Test` z powyższego przykładu może być wykorzystana wraz z wszelkimi modyfikatorami w rodzaju `public` czy `static`. W ujęciu składniowym adnotacje to elementy stojące na równi z takimi właśnie modyfikatorami.

Definiowanie adnotacji

Poniżej przedstawiam definicję adnotacji z ostatniego przykładu. Jak widać, definicja adnotacji przypomina nieco definicję interfejsu. W rzeczy samej definicje adnotacji kompilują się do postaci plików klas — zupełnie jak zwyczajne interfejsy w języku Java:

```
//: net/mindview/atunit/Test.java
// Znacznik @Test.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {} ///:~
```

Jeśli pominąć symbol `@`, to definicja `@Test` nie różni się specjalnie od definicji pustego interfejsu. Definicja adnotacji wymaga również metaadnotacji `@Target` i `@Retention`. `@Target` określa miejsce stosowania adnotacji (na przykład przy metodzie albo polu), a `@Retention` („podtrzymanie") mówi, czy adnotacje mają być dostępne w kodzie źródłowym (`SOURCE`), w plikach klas (`CLASS`) czy może w czasie wykonania (`RUNTIME`).

Adnotacje zazwyczaj zawierają *elementy* określające wartości adnotacji. Program bądź narzędzie może potem wykorzystywać owe elementy przy przetwarzaniu adnotacji. Elementy przypominają metody interfejsu, z tym że pozwalają na deklarowanie wartości domyślnych.

Adnotacja pozbawiona jakichkolwiek elementów, jak powyższa adnotacja `@Test`, to tak zwany *marker* (ang. *marker annotation*).

Oto prosta adnotacja rejestrująca przypadki użycia w projekcie. Programiści opatrują adnotacjami wszystkie metody albo zestawy metod, które spełniają wymagania danego przypadku użycia. Kierownik projektu może potem szacować postępy, zliczając zaimplementowane przypadki użycia, a programiści opiekujący się projektem mogą łatwo wyszukiwać przypadki użycia, w których trzeba dokonać aktualizacji albo diagnostyki reguł biznesowych.

```
//: annotations/UseCase.java
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UseCase {
  public int id();
  public String description() default "brak opisu";
} ///:~
```

Zauważ, że `id` i `description` mocno przypominają deklaracje metod. Ponieważ element `id` podlega kontroli typu ze strony kompilatora, stanowi godny zaufania mechanizm łączenia bazy danych rejestru z opisem przypadku użycia i kodem źródłowym. Element `description` posiada wartość domyślną (`default`) wybieraną przez procesor adnotacji wtedy, kiedy adnotacja opatrująca metodę nie otrzyma jawnie żadnej wartości.

Oto klasa z trzema metodami opatrzonymi adnotacjami kwalifikującymi je jako przypadki użycia:

```
//: annotations/PasswordUtils.java
import java.util.*;

public class PasswordUtils {
  @UseCase(id = 47, description =
  "Hasło musi zawierać przynajmniej jedną cyfrę")
  public boolean validatePassword(String password) {
    return (password.matches("\\w*\\d\\w*"));
  }
  @UseCase(id = 48)
  public String encryptPassword(String password) {
   return new StringBuilder(password).reverse().toString();
  }
  @UseCase(id = 49, description =
  "Nowe hasło nie może być identyczne z poprzednimi")
  public boolean checkForNewPassword(
    List<String> prevPasswords, String password) {
    return !prevPasswords.contains(password);
  }
} ///:~
```

Wartości elementów adnotacji są wyrażane parami nazwa-wartość umieszczonymi w nawiasie za deklaracją `@UseCase`. Adnotacja dla metody `encryptPassword()` nie zawiera wartości dla elementu `description`, więc kiedy klasa `PasswordUtils` będzie przetwarzana przez procesor adnotacji, w `@interface UseCase` wybrana zostanie wartość domyślna.

Wyobraź sobie wykorzystanie podobnego systemu do „naszkicowania" systemu, a potem wypełniania go właściwą treścią, w miarę implementacji.

Metaadnotacje

Obecnie wyróżniono jedynie trzy adnotacje predefiniowane (opisywane wcześniej) i cztery metaadnotacje zdefiniowane w samym języku Java. Metaadnotacje to adnotacje dla adnotacji:

`@Target`	Miejsce stosowania adnotacji. Możliwe argumenty `ElementType` to:
	`CONSTRUCTOR` — w deklaracji konstruktora
	`FIELD` — w deklaracji pola (w tym w deklaracjach wartości stałych wyliczeniowych)
	`LOCAL_VARIABLE` — w deklaracji zmiennej lokalnej
	`METHOD` — w deklaracji metody
	`PACKAGE` — w deklaracji pakietu
	`PARAMETER` — w deklaracji parametru
	`TYPE` — w deklaracji klasy, interfejsu (w tym typu adnotacji) bądź typu wyliczeniowego.
`@Retention`	Okres trwałości adnotacji. Możliwe argumenty `RetentionPolicy` to:
	`SOURCE` — adnotacje unieważniane przez kompilator
	`CLASS` — adnotacje widoczne w plikach klas, unieważniane przez maszynę wirtualną
	`RUNTIME` — adnotacje podtrzymywane w maszynie wirtualnej w czasie wykonania, a więc nadające się do odczytu za pośrednictwem refleksji.
`@Documented`	Ujęcie danej adnotacji w dokumentacji Javadoc.
`@Inherited`	Zezwolenie na dziedziczenie adnotacji w podklasach.

Przez większość czasu będziesz definiował własne adnotacje i pisał własne procesory do ich przetwarzania.

Procesory adnotacji

Same adnotacje, bez narzędzi umożliwiających ich odczytywanie i przetwarzanie, byłyby niewiele przydatniejsze od najzwyklejszych komentarzy. Ważnym elementem procesu stosowania adnotacji jest tworzenie *procesora adnotacji*. Otóż Java SE5 udostępnia rozszerzenie interfejsu refleksji przeznaczone właśnie do tworzenia wspomnianych narzędzi. Dodatkowo programista ma do dyspozycji zewnętrzne narzędzie `apt`, pomocne w analizie leksykalnej kodu źródłowego z adnotacjami.

Poniżej przedstawiam bardzo prosty procesor adnotacji odczytujący opatrzoną adnotacją klasę `PasswordUtils` i wykorzystujący refleksję do odszukania znaczników `@UseCase`. Na bazie listy identyfikatorów `id` program ten konstruuje listę odnalezionych przypadków użycia i listę braków:

```
//: annotations/UseCaseTracker.java
import java.lang.reflect.*;
import java.util.*;

public class UseCaseTracker {
  public static void
  trackUseCases(List<Integer> useCases, Class<?> cl) {
    for(Method m : cl.getDeclaredMethods()) {
      UseCase uc = m.getAnnotation(UseCase.class);
      if(uc != null) {
        System.out.println("Odnaleziony przypadek użycia:" + uc.id() +
          " " + uc.description());
        useCases.remove(new Integer(uc.id()));
      }
    }
    for(int i : useCases) {
      System.out.println("Ostrzeżenie: brak przypadku użycia " + i);
    }
  }
  public static void main(String[] args) {
    List<Integer> useCases = new ArrayList<Integer>();
    Collections.addAll(useCases, 47, 48, 49, 50);
    trackUseCases(useCases, PasswordUtils.class);
  }
} /* Output:
Odnaleziony przypadek użycia:47 Hasło musi zawierać przynajmniej jedną cyfrę
Odnaleziony przypadek użycia:48 brak opisu
Odnaleziony przypadek użycia:49 Nowe hasło nie może być identyczne z poprzednimi
Ostrzeżenie: brak przypadku użycia-50
*///:~
```

Program wykorzystuje metodę refleksji `getDeclaredMethods()` i metodę `getAnnotation()` pochodzącą z interfejsu `AnnotatedElement` (interfejs ten implementują klasy takie jak `Class`, `Field` czy `Method`). Metoda ta zwraca obiekt adnotacji określonego typu, w tym przypadku typu `UseCase`. Gdyby metoda nie posiadała żadnych adnotacji danego typu,

wywołanie zwróciłoby wartość `null`. Wartości elementów adnotacji wyłuskuje się za pośrednictwem wywołań `id()` i `description()`. Pamiętaj, że w adnotacji dla metody `encryptPassword()` zabrakło wartości dla elementu `description`, więc procesor odnajduje w tym przypadku wartość domyślną `"Brak opisu"`.

Elementy adnotacji

Znacznik `@UseCase` zdefiniowany w programie *UseCase.java* zawiera element `id` typu `int` i element `description` typu `String`. Oto lista dozwolonych typów elementów adnotacji:

- wszystkie typy podstawowe (`int`, `float`, `boolean` itp.),
- typ `String`,
- typ `Class`,
- typy wyliczeniowe,
- adnotacje,
- tablice elementów powyższych typów.

Jeśli spróbujesz zadeklarować element innego typu, kompilator oprotestuje próbę komunikatem o błędzie. Zauważ, że nie możesz zastosować żadnej z klas opakowujących, ale istnieje mechanizm automatycznego pakowania wartości podstawowych w obiekty, więc nie jest to żadnym problemem. Elementy adnotacji mogą też być same w sobie adnotacjami. Przekonasz się wkrótce, że zagnieżdżanie adnotacji może być wielce użyteczne.

Ograniczenia wartości domyślnych

Kompilator jest dość wybredny odnośnie domyślnych wartości elementów. Żaden element nie może pozostać bez wartości. Oznacza to, że elementy muszą otrzymywać wartości w miejscu użycia adnotacji albo posiadać wartości domyślne.

Istnieje jeszcze jedno ograniczenie, w postaci wymogu, aby żaden z elementów typów podstawowego nie otrzymywał wartości `null`, niezależnie od tego, czy jest to wartość nadawana w kodzie źródłowym czy wartość domyślna określona w definicji interfejsu adnotacji. Utrudnia to pisanie procesora, który bazuje na obecności albo braku któregoś z elementów, bo zasadniczo każdy element adnotacji musi być obecny w każdym jej wystąpieniu. Trzeba więc w takich przypadkach wyróżniać wartości reprezentujące brak, na przykład przez ciągi puste bądź liczby ujemne:

```
//: annotations/SimulatingNull.java
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SimulatingNull {
  public int id() default -1;
  public String description() default "";
} ///:~
```

To dość typowe podejście do definiowania adnotacji.

Generowanie plików zewnętrznych

Adnotacje są szczególnie użyteczne w pracy ze szkieletami aplikacji, w których kod źródłowy powinien być uzupełniany pewnego rodzaju informacjami zewnętrznymi. Przykładem jest technologia Enterprise Java Beans (w wersjach sprzed EJB3), wymagająca wielu różnych interfejsów i deskryptorów wdrożenia, elementów szablonowych — definiowanych tak samo dla każdego komponentu EJB — ale niezbędnych. Z kolei usługi WWW, biblioteki znaczników własnych i narzędzia odwzorowania obiektowo-relacyjnego, jak Toplink czy Hibernate, często wymagają deskryptorów XML zewnętrznych wobec kodu źródłowego. Po zdefiniowaniu klasy języka Java programista musi przejść znój żmudnego powtarzania informacji w rodzaju nazwy, pakietu i tak dalej — mimo że komplet tych informacji występuje w samej klasie. Kiedy w użyciu są zewnętrzne pliki deskryptorów, dochodzi do wydzielenia dwóch źródeł informacji o klasie, co często skutkuje problemami z synchronizowaniem kodu. Do tego programista pracujący nad projektem musi nie tylko posiadać umiejętność programowania, ale również opanować sztukę pisania deskryptorów.

Załóżmy, że chcemy udostępnić proste odwzorowanie obiektowo-relacyjne w celu automatyzacji tworzenia tabeli bazy danych mającej przechowywać komponent JavaBean. Do określania nazwy klasy, kompletu składowych i informacji o ich odwzorowaniu w bazie danych moglibyśmy wykorzystać plik deskryptora w języku XML, ale dzięki adnotacjom możemy wszystkie te informacje skupić w pliku kodu źródłowego klasy komponentu JavaBean. Potrzebujemy adnotacji definiującej nazwę tabeli bazy danych skojarzonej z komponentem, nazwy kolumn i określenia typów SQL dla poszczególnych właściwości komponentu.

Oto przykład adnotacji dla komponentu instruującej procesor adnotacji co do tworzenia tabeli bazy danych:

```
//: annotations/database/DBTable.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.TYPE) // Dotyczy wyłącznie klas
@Retention(RetentionPolicy.RUNTIME)
public @interface DBTable {
  public String name() default "";
} ///:~
```

Każdy `ElementType` określany w adnotacji `@Target` to ograniczenie powiadamiające kompilator, że dana adnotacja może być stosowana jedynie do podanego typu. Można podać pojedynczą wartość typu wyliczeniowego `ElementType` albo listę dowolnych kombinacji wartości (oddzielanych przecinkami). Jeśli adnotacja ma się nadawać do opatrywania dowolnych typów `ElementType`, można całkiem zrezygnować z adnotacji `@Target`, choć to dość rzadki przypadek.

Zauważ, że definicja adnotacji `@DBTable` posiada element `name()`, tak aby w miejscu użycia adnotacji można było podać nazwę tabeli do utworzenia przez procesor adnotacji:

Oto adnotacje dla pól komponentu JavaBean:

```
//: annotations/database/Constraints.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Constraints {
  boolean primaryKey() default false;
  boolean allowNull() default true;
  boolean unique() default false;
} ///:~

//: annotations/database/SQLString.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLString {
  int value() default 0;
  String name() default "";
  Constraints constraints() default @Constraints;
} ///:~

//: annotations/database/SQLInteger.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLInteger {
  String name() default "";
  Constraints constraints() default @Constraints;
} ///:~
```

Adnotacja `Constraints` pozwala procesorowi na wyłuskanie metadanych o tabeli bazy danych. Są one reprezentowane niewielkim podzbiorem więzów integralnościowych typowych dla baz danych i jako elementy poglądowe powinny wystarczyć. Elementy `primaryKey()`, `allowNull()` i `unique()` otrzymały sensowne wartości domyślne, tak aby użytkownik nie musiał przy stosowaniu adnotacji zbyt wiele pisać.

Pozostałe dwa interfejsy adnotacji definiują typy SQL. Aby przykład miał jakąkolwiek użyteczność, trzeba by zdefiniować adnotacje dla wszystkich typów SQL. Tu dwa typy wystarczą.

Rzeczone typy posiadają elementy `name()` i `constraints()`. Ten ostatni korzysta z możliwości zagnieżdżania adnotacji, osadzając w adnotacji informacje o więzach integralnościowych danej kolumny w tabeli bazodanowej. Zauważ, że domyślna wartość dla elementu `constraints()` to `@Constraints`. Z racji braku wartości elementów w (również nieobecnym) nawiasie, domyślną wartością elementu `constraints()` jest tu adnotacja `@Constraint` z domyślnym zestawem wartości. Aby w zagnieżdżonej adnotacji `@Constraints` wymusić atrybut niepowtarzalności wartości w kolumnie, należałoby zdefiniować element `constraints()` jak poniżej:

```
//: annotations/database/Uniqueness.java
// Próbka zagnieżdżania adnotacji
package annotations.database;

public @interface Uniqueness {
  Constraints constraints()
    default @Constraints(unique=true);
} ///:~
```

A oto prosty komponent JavaBean wykorzystujący powyższe adnotacje:

```
//: annotations/database/Member.java
package annotations.database;

@DBTable(name = "MEMBER")
public class Member {
  @SQLString(30) String firstName;
  @SQLString(50) String lastName;
  @SQLInteger Integer age;
  @SQLString(value = 30,
  constraints = @Constraints(primaryKey = true))
  String handle;
  static int memberCount;
  public String getHandle() { return handle; }
  public String getFirstName() { return firstName; }
  public String getLastName() { return lastName; }
  public String toString() { return handle; }
  public Integer getAge() { return age; }
} ///:~
```

Adnotacja klasy `@DBTable` otrzymała wartość `MEMBER`, która zostanie wykorzystana w roli nazwy tabeli w bazie danych. Właściwości komponentu, w postaci pól `firstName` i `lastName`, zostały opatrzone adnotacjami `@SQLString` z wartościami elementów (odpowiednio) 30 i 50. Adnotacje te są interesujące z dwóch względów: po pierwsze, wykorzystują domyślną wartość zagnieżdżonej adnotacji `@Constraints`, po drugie zaś, wykorzystują pewien skrót. Otóż jeśli element adnotacji ma nazwę `value`, to dopóki jest on jedynym elementem otrzymującym wartość w adnotacji, przy nadawaniu tej wartości nie trzeba stosować składni nazwa-wartość; wystarczy podać wartość w nawiasie. Możliwość ta dotyczy wszystkich dozwolonych typów elementów. Pewne ograniczenie użyteczności skrótu, wynikające z konieczności stosowania nazwy `value` dla elementu, jest rekompensowane prostotą zapisu adnotacji i przejrzystością semantyczną:

```
@SQLString(30)
```

Procesor adnotacji wykorzysta tę wartość do ustawienia rozmiaru tworzonej wartości w kolumnie tabeli SQL.

Wygoda stosowania wartości domyślnych szybko okazuje się utrapieniem. Spójrz choćby na adnotację dla pola `handle`. To adnotacja `@SQLString`, ale określająca klucz główny tabeli, co wymaga jawnego ustawienia elementu `primaryKey` zagnieżdżonej adnotacji `@Constraints`. I tu zaczynają się komplikacje. Trzeba uciec się do rozwlekłej składni par nazwa-wartość, z powtórzeniami nazw elementów, oraz nazwy adnotacji zagnieżdżonej (tu `@Constraints`). Ponieważ zaś w adnotacji obejmującej element `value` przestaje być jedynym elementem otrzymującym wartość, nie można skorzystać z zapisu skróconego. Efekt końcowy jest mało elegancki.

Rozwiązania alternatywne

Adnotacje dla postawionego zadania można też utworzyć na inne sposoby. Można by, na przykład, wydzielić pojedynczą klasę adnotacji o nazwie `@TableColumn` z elementem typu wyliczeniowego, definiującego wartości `STRING`, `INTEGER`, `FLOAT` i tak dalej. Eliminuje to konieczność tworzenia adnotacji dla każdego typu SQL z osobna, za to uniemożliwia kwalifikowanie typów dodatkowymi elementami, jak *rozmiar* czy *precyzja*, co niekiedy jest wielce przydatne.

Do opisu typu SQL można by też użyć elementu typu `String` i nadawać mu wartości takie jak „VARCHAR(30)" czy „INTEGER". Pozwalałoby to na kwalifikowanie typów, ale wiązałoby odwzorowanie typu Java do typu SQL w kodzie, co jest niepożądane. Zmiana baz danych nie powinna wymuszać ponownej kompilacji kodu aplikacji; elegantsze rozwiązanie polegałoby na poinstruowaniu procesora adnotacji o stosowaniu „odmiany" SQL i zdanie się na definiowanie konkretnych typów SQL przy przetwarzaniu adnotacji.

Trzecie rozwiązanie wymagałoby użycia w adnotacji danego pola dwóch typów adnotacji w połączeniu: `@Constraints` i odpowiedniego typu SQL (np. `@SQLInteger`). Byłoby to nieco zagmatwane, ale kompilator nie ogranicza nijak liczby adnotacji skojarzonej z danym elementem kodu; tyle że przy stosowaniu wielu adnotacji nie wolno ich powtarzać.

Adnotacje nie dają się dziedziczyć

W definicji adnotacji nie można stosować słowa kluczowego `extends`. Szkoda, bo najbardziej eleganckim rozwiązaniem rozważanego problemu byłoby zdefiniowanie adnotacji `@TableColumn` z zagnieżdżoną adnotacją `@SQLType`. Można by wtedy wyprowadzać wszystkie typy SQL (`@SQLString` czy `@SQLInteger`) z typu `@SQLType`. Zredukowałoby to też zawiłości składni oznaczania adnotacjami. Na razie jednak nie zanosi się na możliwość dziedziczenia adnotacji, więc póki co należałoby ograniczyć się do wymienionych alternatyw.

Implementowanie procesora

Oto przykład procesora adnotacji, który wczytuje plik klasy, sprawdza adnotacje dotyczące bazy danych i generuje polecenie SQL zakładające tabele bazy danych:

```
// : annotations/database/TableCreator.java
// Procesor adnotacji na bazie refleksji.
// {Args: annotations.database.Member}
package annotations.database;
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.util.*;

public class TableCreator {
  public static void main(String[] args) throws Exception {
    if(args.length < 1) {
      System.out.println("argumenty: klasy z adnotacjami");
      System.exit(0);
    }
```

```
    for(String className : args) {
      Class<?> cl = Class.forName(className);
      DBTable dbTable = cl.getAnnotation(DBTable.class);
      if(dbTable == null) {
        System.out.println(
          "Brak adnotacji DBTable dla klasy " + className);
        continue;
      }
      String tableName = dbTable.name();
      // Jeśli nazwa jest pusta, używamy nazwy klasy:
      if(tableName.length() < 1)
        tableName = cl.getName().toUpperCase();
      List<String> columnDefs = new ArrayList<String>();
      for(Field field : cl.getDeclaredFields()) {
        String columnName = null;
        Annotation[] anns = field.getDeclaredAnnotations();
        if(anns.length < 1)
          continue; // Pole nie jest odwzorowywane w bazie danych
        if(anns[0] instanceof SQLInteger) {
          SQLInteger sInt = (SQLInteger) anns[0];
          // W przypadku braku nazwy kolumny używamy nazwy pola.
          if(sInt.name().length() < 1)
            columnName = field.getName().toUpperCase();
          else
            columnName = sInt.name();
          columnDefs.add(columnName + " INT" +
            getConstraints(sInt.constraints()));
        }
        if(anns[0] instanceof SQLString) {
          SQLString sString = (SQLString) anns[0];
          // W przypadku braku nazwy kolumny używamy nazwy pola.
          if(sString.name().length() < 1)
            columnName = field.getName().toUpperCase();
          else
            columnName = sString.name();
          columnDefs.add(columnName + " VARCHAR(" +
            sString.value() + ")" +
            getConstraints(sString.constraints()));
        }
        StringBuilder createCommand = new StringBuilder(
          "CREATE TABLE " + tableName + "(");
        for(String columnDef : columnDefs)
          createCommand.append("\n    " + columnDef + ",");
        // Usunięcie średnika z końca
        String tableCreate = createCommand.substring(
          0, createCommand.length() - 1) + ");";
        System.out.println("Tworzenie tabeli SQL dla klasy " +
          className + ":\n" + tableCreate);
      }
    }
  }
  private static String getConstraints(Constraints con) {
    String constraints = "";
    if(!con.allowNull())
      constraints += " NOT NULL";
    if(con.primaryKey())
      constraints += " PRIMARY KEY";
```

```
      if(con.unique())
        constraints += " UNIQUE";
      return constraints;
    }
} /* Output:
Tworzenie tabeli SQL dla klasy annotations.database.Member:
CREATE TABLE MEMBER(
  FIRSTNAME VARCHAR(30));
Tworzenie tabeli SQL dla klasy annotations.database.Member:
CREATE TABLE MEMBER(
  FIRSTNAME VARCHAR(30),
  LASTNAME VARCHAR(50));
Tworzenie tabeli SQL dla klasy annotations.database.Member:
CREATE TABLE MEMBER(
  FIRSTNAME VARCHAR(30),
  LASTNAME VARCHAR(50),
  AGE INT);
Tworzenie tabeli SQL dla klasy annotations.database.Member:
CREATE TABLE MEMBER(
  FIRSTNAME VARCHAR(30),
  LASTNAME VARCHAR(50),
  AGE INT,
  HANDLE VARCHAR(30) PRIMARY KEY);
*///:~
```

Metoda `main()` przegląda nazwy kolejnych klas występujących w wierszu wywołania programu. Każda z tych klas jest ładowana za pomocą metody `forName()`; w załadowanej klasie za pośrednictwem wywołania `getAnnotation(DBTable.class)` wyszukiwana jest adnotacja `@DBTable`. Jeśli taka adnotacja się znajdzie, odszukiwana jest nazwa tabeli bazy danych. Wtedy metoda `main()` przechodzi do wczytywania kolejnych pól klasy i wykrywania adnotacji zadeklarowanych dla tej klasy (`getDeclaredAnnotations()`). Owocuje to tablicą adnotacji zdefiniowanych dla danego pola tudzież metody. Operator `instanceof` służy tu do ustalenia, czy owe adnotacje są typu `@SQLInteger` bądź `@SQLString`, dla których to następuje utworzenie fragmentu zapytania SQL z odpowiednią nazwą kolumny i typem danych. Zauważ, że z braku dziedziczenia interfejsów adnotacji wywołanie `getDeclaredAnnotations()` to jedyny sposób przybliżenia zachowania polimorficznego.

Do metody `getConstraints()` przekazywana jest zagnieżdżona adnotacja `@Constraint`; metoda `getConstraint()` konstruuje fragment ciągu zapytania z więzami integralnościowymi danego pola.

Warto zaznaczyć, że zaprezentowana technika to jednak nieco naiwny sposób definiowania odwzorowania obiektowo-relacyjnego. Obecność adnotacji `@DBTable` przyjmującej nazwę tabeli wymusza ponowną kompilację kodu za każdym razem, kiedy zechcemy zmienić strukturę relacyjną w bazie danych. Zwykle jest to niepożądane. Istnieje zresztą wiele gotowych infrastruktur odwzorowania obiektów na tabele baz danych; coraz więcej spośród nich korzysta również z adnotacji.

Ćwiczenie 1. Zaimplementuj w przykładzie odwzorowania obiektowo-relacyjnego dodatkowe typy SQL (2).

Projekt[3]**.** Zmodyfikuj przykład z odwzorowaniem obiektowo-relacyjnym tak, aby faktycznie nawiązywał połączenie z prawdziwą bazą danych i manipulował jej schematem za pośrednictwem interfejsu JDBC.

Projekt. Zmodyfikuj przykład z odwzorowaniem obiektowo-relacyjnym tak, aby zamiast odwoływać się do bazy danych, tworzył dokumenty XML.

Przetwarzanie adnotacji za pomocą apt

Narzędzie apt (*annotations processing tool*) to pierwsza wersja narzędzia firmy Sun służącego do wspomagania zadania przetwarzania adnotacji. Ponieważ to jego wczesne wcielenie, można mu wiele zarzucić, jak choćby to, że jest nieco prymitywne, ale posiada kilka funkcji ułatwiających życie programisty.

Podobnie jak javac, program apt jest przeznaczony do przetwarzania plików kodu źródłowego (nie plików skompilowanych klas). Domyślnie apt po przetworzeniu adnotacji występujących w kodzie źródłowym kompiluje pliki kodu źródłowego. Można to wykorzystać do automatycznego tworzenia nowych plików kodu źródłowego w ramach procesu kompilacji. W rzeczy samej, w tym samym przebiegu apt przegląda nowo utworzone pliki pod kątem adnotacji i podejmuje ich kompilację — to bardzo wygodne.

Jeśli procesor adnotacji tworzy nowy plik kodu źródłowego, plik ten jest również sprawdzany pod kątem obecności adnotacji w ramach kolejnej *rundy* (tak określa się to w dokumentacji) przetwarzania. Narzędzie kontynuuje przetwarzanie runda po rundzie, aż do momentu, w którym w wyniku przetwarzania nie dojdzie do utworzenia nowych plików kodu źródłowego. Potem wszystkie zebrane pliki źródłowe są kompilowane.

Każda adnotacja napisana przez programistę wymaga własnego procesora, ale narzędzie apt potrafi grupować procesory adnotacji. Dzięki temu można „za jednym zamachem" przetwarzać wiele klas, co jest znacznie wygodniejsze niż samodzielne przeglądanie wielu obiektów `File`. Programista może też tworzyć moduły oczekujące na powiadomienia o zakończeniu bieżącej rundy przetwarzania.

W czasie pisania niniejszej książki narzędzie apt nie było jeszcze dostępne w postaci zadania systemu automatycznej kompilacji Ant (zobacz suplement *http://MindView.net/Books/BetterJava*), ale mogło być uruchomione jako zewnętrzne zadanie z poziomu Ant. Aby skompilować procesory adnotacji prezentowane w tym rozdziale, trzeba umieścić w zasięgu ścieżki `CLASSPATH` pakiet *tools.jar*; zawiera on między innymi interfejsy `com.sun.mirror.*`.

[3] Proponowane projekty można wykorzystać na przykład jako warunek zaliczenia semestru. Rozwiązania dostępne dla zwykłych ćwiczeń nie zawierają oczywiście rozwiązań projektów.

Program apt wykorzystuje wytwórnię `AnnotationProcessorFactory`, za pomocą której dla każdej znalezionej adnotacji tworzy odpowiedni procesor adnotacji. Przy uruchamianiu programu apt należy podać klasę wytwórni albo zadbać o umieszczenie potrzebnych wytwórni w zasięgu ścieżki klas. Jeśli się tego zaniedba, apt uruchomi proces *wykrywania*, którego szczegóły zostały opisane w dokumentacji firmy Sun, w rozdziale *Developing an Annotation Processor*.

Przygotowując własny procesor adnotacji na użytek apt, nie możemy korzystać z interfejsu refleksji, bo będziemy operować na kodzie źródłowym, a nie klasach skompilowanych[4]. Problem eliminuje dodatkowy interfejs programistyczny `mirror`[5] pozwalający na podglądanie metod, pól i typów w nieskompilowanym kodzie źródłowym.

Oto adnotacja, którą moglibyśmy wykorzystać do wyłuskania metod publicznych z klasy i ich konwersji na postać interfejsu:

```
//: annotations/ExtractInterface.java
// Przetwarzanie adnotacji na bazie APT.
package annotations;
import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface ExtractInterface {
  public String value();
} ///:~
```

`RetentionPolicy` adnotacji `ExtractInterface` ma tu wartość `SOURCE`, bo obecność adnotacji po wyodrębnieniu interfejsu z klasy jest zbędna. Poniższa klasa udostępnia metodę publiczną, na której będziemy mogli przetestować przetwarzanie adnotacji wyodrębniającej interfejs:

```
//: annotations/Multiplier.java
// Przetwarzanie adnotacji na bazie APT.
package annotations;

@ExtractInterface("IMultiplier")
public class Multiplier {
  public int multiply(int x, int y) {
    int total = 0;
    for(int i = 0; i < x; i++)
      total = add(total, y);
    return total;
  }
  private int add(int x, int y) { return x + y; }
  public static void main(String[] args) {
    Multiplier m = new Multiplier();
    System.out.println("11*16 = " + m.multiply(11, 16));
  }
} /* Output:
11*16 = 176
*///:~
```

[4] Chyba że użyjemy niestandardowej opcji `-XclassesAsDecls` — wtedy możemy operować na adnotacjach w skompilowanych plikach klas.

[5] Projektanci języka Java używają tu gry słów: w lustrze (*mirror*) widać odbicie (*reflection*).

Klasa `Multiplier` (nawiasem mówiąc, operująca wyłącznie na nieujemnych liczbach całkowitych) posiada metodę `multiply()`, która realizuje operację mnożenia za pomocą wielokrotnych wywołań prywatnej metody `add()`. Metoda `add()` jest niepubliczna, więc nie stanowi elementu interfejsu. Adnotacja klasy otrzymała wartość `IMultiplier` określającą nazwę interfejsu, który ma być wyodrębniony z klasy.

Teraz potrzebujemy procesora, który zrealizuje zadanie wyodrębniania interfejsu:

```
//: annotations/InterfaceExtractorProcessor.java
// Przetwarzanie adnotacji na bazie APT.
// {Exec: apt -factory
// annotations.InterfaceExtractorProcessorFactory
// Multiplier.java -s ../annotations}
package annotations;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.io.*;
import java.util.*;

public class InterfaceExtractorProcessor
  implements AnnotationProcessor {
  private final AnnotationProcessorEnvironment env;
  private ArrayList<MethodDeclaration> interfaceMethods =
    new ArrayList<MethodDeclaration>();
  public InterfaceExtractorProcessor(
    AnnotationProcessorEnvironment env) { this.env = env; }
  public void process() {
    for(TypeDeclaration typeDecl :
      env.getSpecifiedTypeDeclarations()) {
      ExtractInterface annot =
        typeDecl.getAnnotation(ExtractInterface.class);
      if(annot == null)
        break;
      for(MethodDeclaration m : typeDecl.getMethods())
        if(m.getModifiers().contains(Modifier.PUBLIC) &&
           !(m.getModifiers().contains(Modifier.STATIC)))
          interfaceMethods.add(m);
      if(interfaceMethods.size() > 0) {
        try {
          PrintWriter writer =
            env.getFiler().createSourceFile(annot.value());
          writer.println("package " +
            typeDecl.getPackage().getQualifiedName() + ";");
          writer.println("public interface " +
            annot.value() + " {");
          for(MethodDeclaration m : interfaceMethods) {
            writer.print("  public ");
            writer.print(m.getReturnType() + " ");
            writer.print(m.getSimpleName() + " (");
            int i = 0;
            for(ParameterDeclaration parm :
              m.getParameters()) {
              writer.print(parm.getType() + " " +
                parm.getSimpleName());
              if(++i < m.getParameters().size())
                writer.print(", ");
            }
```

```
            writer.println(");");
          }
          writer.println("}");
          writer.close();
        } catch(IOException ioe) {
          throw new RuntimeException(ioe);
        }
      }
    }
  }
} ///:~
```

Główny ciężar przetwarzania przyjmuje metoda `process()`. Klasa `MethodDeclaration` i jej metoda `getModifiers()` służą do zidentyfikowania metod publicznych (za wyjątkiem statycznych metod publicznych) przetwarzanej klasy. Wszelkie znalezione metody są zbierane w kontenerze `ArrayList` i następnie wykorzystywane do wygenerowania metod w definicji nowego interfejsu w nowym pliku *.java*.

Zauważ, że do konstruktora klasy głównej programu przekazywany jest obiekt `AnnotationProcessorEnvironment`. Obiekt ten można wykorzystywać do wyłuskiwania wszelkich typów (definicji klas) przetwarzanych przez apt, a także do pozyskania obiektów `Messager` i `Filer`. Obiekt `Messager` pozwala na wypisywanie komunikatów dla użytkownika, na przykład powiadamianie o umiejscowieniu i charakterze błędów stwierdzonych podczas przetwarzania. Z kolei `Filer` to rodzaj obiektu `PrintWriter` — służy do tworzenia nowych plików. W procesorach adnotacji korzystamy z obiektu Filer zamiast `PrintWriter`, dlatego że `Filer` pozwala programowi apt śledzić nowo tworzone pliki i uwzględniać je w następnych rundach przetwarzania.

Widać też, że metoda `createSourceFile()` otwiera najzwyklejszy strumień wyjściowy z nazwą klasy bądź interfejsu określoną za pośrednictwem wartości adnotacji. Nie ma tu żadnego dodatkowego wsparcia dla tworzenia konstrukcji języka Java, trzeba więc generować kod źródłowy w tym języku za pośrednictwem ogólnych metod `print()` i `println()`. Należy zatem samemu dbać o dopasowanie nawiasów i poprawność składniową generowanego kodu.

Metoda `process()` jest wywoływana przez narzędzie apt, które pozyskuje obiekt procesora z wytwórni takich obiektów:

```
// : annotations/InterfaceExtractorProcessorFactory.java
// Przetwarzanie adnotacji na bazie APT.
package annotations;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.util.*;

public class InterfaceExtractorProcessorFactory
  implements AnnotationProcessorFactory {
  public AnnotationProcessor getProcessorFor(
    Set<AnnotationTypeDeclaration> atds,
    AnnotationProcessorEnvironment env) {
    return new InterfaceExtractorProcessor(env);
  }
  public Collection<String> supportedAnnotationTypes() {
    return
     Collections.singleton("annotations.ExtractInterface");
```

```
  }
  public Collection<String> supportedOptions() {
    return Collections.emptySet();
  }
} ///:~
```

Interfejs `AnnotationProcessorFactory` posiada zaledwie trzy metody. Do pozyskiwania obiektu procesora służy `getProcessorFor()`, przyjmująca kontener `Set` z deklaracjami typu (nazw klas Java, dla których uruchomiono program apt) oraz obiekt `AnnotationProcessorEnvironment`, który jest potem przekazywany do konstruktora odpowiedniego procesora. Pozostałe dwie metody, `supportedAnnotationTypes()` i `supportedOptions()`, służą do sprawdzania, czy wszystkie adnotacje znalezione przez apt są obsługiwane przez odpowiednie procesory i czy obsługiwane są wszystkie opcje wymienione w wierszu wywołania. Szczególnie ważna jest metoda `getProcessorFor()`, bo w przypadku niezwrócenia pełnej nazwy klasy dla typu adnotacji w kolekcji ciągów `String`, apt ostrzeże o braku odpowiedniego procesora i zakończy działanie bez podejmowania przetwarzania.

Procesor i wytwórnia wchodzą w skład pakietu `annotations`; dla założonej struktury katalogów stosowny wiersz polecenia jest osadzony w znaczniku komentarzowym `Exec` na początku pliku *InterfaceExtractorProcessor.java*. Tak wywołany apt jest instruowany odnośnie stosowania klasy zdefiniowanej powyżej i przetwarzania pliku *Multiplier.java*. Opcja `-s` wymusza tworzenie wszelkich nowych plików w katalogu *annotations*. Wygenerowany plik *IMultiplier.java*, którego zawartości można się domyślić, analizując wywołania `println()` w powyższym procesorze, wygląda ostatecznie tak:

```
package annotations;
public interface IMultiplier {
  public int multiply(int x, int y);
}
```

Plik ten również zostanie skompilowany przez program apt, więc w katalogu pojawi się również plik *IMultiplier.class*.

Ćwiczenie 2. Uzupełnij klasę i procesor wyodrębniający interfejs o obsługę dzielenia (metodą `divide()`) (3).

Program apt a wizytacje

Przetwarzanie adnotacji może być procesem mocno skomplikowanym. Powyższy przykład prezentuje stosunkowo prosty procesor, który interpretuje zaledwie jedną adnotację, a mimo to ilość kodu jest już pokaźna. Aby zapobiec windowaniu złożoności przy obsłudze wielu adnotacji i większej liczbie procesorów, można skorzystać z usług interfejsu `mirror`, który udostępnia klasy umożliwiające wykorzystanie wzorca projektowego *Visitor* (wizytator). Wzorzec ten to jeden z klasycznych wzorców projektowych zaproponowanych w książce *Design Patterns*[6] i opisywanego szerzej między innymi w *Thinking in Patterns*.

[6] Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1994 — *przyp. tłum.*

Wzorzec ten zakłada przeglądanie struktury danych bądź kolekcji i realizację pewnej operacji dla każdego z obiektów takiej kolekcji. Kolekcja nie musi być uporządkowana, a operacje wykonywane na poszczególnych obiektach są zależne od ich typu. W ten sposób można rozdzielić operacje od samych obiektów, a więc łatwo dodawać nowe operacje bez konieczności uzupełniania definicji klas obiektów o kolejne metody.

Model ten jest przydatny w przetwarzaniu adnotacji, bo klasa języka Java może być reprezentowana właśnie kolekcją obiektów, takich jak `TypeDeclaration` (deklaracja typu), `FieldDeclaration` (deklaracja pola), `MethodDeclaration` (deklaracja metody) i tak dalej. Wdrażając wzorzec projektowy wizytacji, trzeba udostępnić klasę wizytatora z metodami obsługi poszczególnych typów wizytowanych deklaracji. W ten sposób można zróżnicować przebieg wizytacji dla adnotacji metod, klas, pól i tak dalej.

Wróćmy do przykładu generującego tabelę SQL; tym razem skorzystamy z wytwórni procesorów i procesora korzystającego z wzorca projektowego *Visitor*:

```
// : annotations/database/TableCreationProcessorFactory.java
// Tym razem z użyciem wzorca projektowego Visitor.
// {Exec: apt -factory
// annotations.database.TableCreationProcessorFactory
// database/Member.java -s database}
package annotations.database;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import com.sun.mirror.util.*;
import java.util.*;
import static com.sun.mirror.util.DeclarationVisitors.*;

public class TableCreationProcessorFactory
  implements AnnotationProcessorFactory {
  public AnnotationProcessor getProcessorFor(
    Set<AnnotationTypeDeclaration> atds,
    AnnotationProcessorEnvironment env) {
    return new TableCreationProcessor(env);
  }
  public Collection<String> supportedAnnotationTypes() {
    return Arrays.asList(
      "annotations.database.DBTable",
      "annotations.database.Constraints",
      "annotations.database.SQLString",
      "annotations.database.SQLInteger");
  }
  public Collection<String> supportedOptions() {
    return Collections.emptySet();
  }
  private static class TableCreationProcessor
    implements AnnotationProcessor {
    private final AnnotationProcessorEnvironment env;
    private String sql = "";
    public TableCreationProcessor(
      AnnotationProcessorEnvironment env) {
      this.env = env;
    }
    public void process() {
      for(TypeDeclaration typeDecl :
        env.getSpecifiedTypeDeclarations()) {
        typeDecl.accept(getDeclarationScanner(
```

```
            new TableCreationVisitor(), NO_OP));
        sql = sql.substring(0, sql.length() - 1) + ");";
        System.out.println("Generowane zapytanie SQL:\n" + sql);
        sql = "";
      }
    }
    private class TableCreationVisitor
      extends SimpleDeclarationVisitor {
      public void visitClassDeclaration(
        ClassDeclaration d) {
        DBTable dbTable = d.getAnnotation(DBTable.class);
        if(dbTable != null) {
          sql += "CREATE TABLE ";
          sql += (dbTable.name().length() < 1)
            ? d.getSimpleName().toUpperCase()
            : dbTable.name();
          sql += " (";
        }
      }
      public void visitFieldDeclaration(
        FieldDeclaration d) {
        String columnName = "";
        if(d.getAnnotation(SQLInteger.class) != null) {
          SQLInteger sInt = d.getAnnotation(
              SQLInteger.class);
          // W przypadku braku nazwy kolumny używamy nazwy pola.
          if(sInt.name().length() < 1)
            columnName = d.getSimpleName().toUpperCase();
          else
            columnName = sInt.name();
          sql += "\n    " + columnName + " INT" +
            getConstraints(sInt.constraints()) + ",";
        }
        if(d.getAnnotation(SQLString.class) != null) {
          SQLString sString = d.getAnnotation(
              SQLString.class);
          // W przypadku braku nazwy kolumny używamy nazwy pola.
          if(sString.name().length() < 1)
            columnName = d.getSimpleName().toUpperCase();
          else
            columnName = sString.name();
          sql += "\n    " + columnName + " VARCHAR(" +
            sString.value() + ")" +
            getConstraints(sString.constraints()) + ",";
        }
      }
      private String getConstraints(Constraints con) {
        String constraints = "";
        if(!con.allowNull())
          constraints += " NOT NULL";
        if(con.primaryKey())
          constraints += " PRIMARY KEY";
        if(con.unique())
          constraints += " UNIQUE";
        return constraints;
      }
    }
  }
} ///:~
```

Wynik przetwarzania adnotacji powinien być podobny jak w pierwotnym przykładzie `DBTable`.

Procesor i wizytator są w tym przykładzie klasami wewnętrznymi. Zauważ, że metoda `process()` jedynie dodaje klasę wizytatora i inicjalizuje ciąg zapytania SQL.

Oba parametry metody `getDeclarationScanner()` to wizytatory; pierwszy jest wykorzystywany przed, a drugi po wizytacji poszczególnych deklaracji. Ten procesor wymaga jedynie pierwszej wizytacji, więc w miejsce drugiego parametru przekazaliśmy wartość `NO_OP`. Wartość ta odnosi się do statycznego pola interfejsu `DeclarationVisitor`, które zawiera wizytatora-atrapę (powstrzymującego się od jakichkolwiek działań).

`TableCreationVisitor` rozszerza klasę `SimpleDeclarationVisitor`, przesłaniając dwie metody: `visitClassDeclaration()` i `visitFieldDeclaration()`. Klasa `SimpleDeclarationVisitor` to adapter implementujący komplet metod interfejsu `DeclarationVisitor`, możesz więc skoncentrować się na tych, których faktycznie potrzebujesz. W metodzie `visitClassDeclaration()` następuje przejrzenie obiektu `ClassDeclaration` pod kątem adnotacji `@DBTable`; jeśli taka adnotacja występuje, inicjalizowana jest pierwsza część ciągu zapytania SQL. W metodzie `visitFieldDeclaration()` następuje przeszukanie adnotacji pól i wyodrębnienie informacji o typach SQL, podobnie jak to miało miejsce w pierwotnej wersji przykładu.

Wydaje się, że to sposób bardziej skomplikowany, ale jego zaletą jest niezła skalowalność. W miarę wzrostu złożoności procesora adnotacji całkowicie samodzielna implementacja procesora, jak we wcześniejszym przykładzie, może okazać się nazbyt skomplikowana.

Ćwiczenie 3. Uzupełnij program *TableCreationProcessorFactory.java* o dodatkowe typy danych SQL (2).

Adnotacje w testowaniu jednostkowym

Testowanie jednostkowe (ang. *unit testing*) to praktyka tworzenia pewnej liczby testów dla każdej z metod klasy; testy te mają upewnić programistę co do poprawności zachowania wszystkich części klasy. Najpopularniejszym narzędziem realizacji takich testów w Javie jest *JUnit*; w czasie przygotowywania niniejszej książki JUnit doczekał się aktualizacji do wersji opatrzonej numerem 4, przystosowanej do adnotacji[7]. Jedną z uciążliwości stosowania narzędzia JUnit w wersjach pozbawionych obsługi adnotacji był „ceremoniał" konieczny do ustalenia i uruchomienia testów JUnit. Z czasem ów narzut stopniowo się kurczył, ale dopiero adnotacje przybliżyły testowanie do ideału „najprostszego możliwego systemu testującego".

[7] Zastanawiałem się sam nad napisaniem własnego wariantu JUnit z adnotacjami, z wykorzystaniem omawianych tu technik, ale wraz z wydaniem JUnit4, w którym wcielono znaczną część prezentowanych koncepcji, pomysł upadł.

Owe wcześniejsze wersje JUnit wymagały tworzenia osobnej klasy przechowującej testy. Dzięki adnotacjom możemy te testy osadzić w testowanej klasie i tym samym do minimum zmniejszyć wysiłek i czas poświęcany testowaniu. Dodatkową korzyścią jest możliwość testowania metod prywatnych równie łatwo jak metod publicznych.

Ponieważ poniższa, przykładowa infrastruktura testowa bazuje na adnotacjach, nosi nazwę @Unit (*AtUnit*). Najprostsza i najczęściej stosowana forma testowania wymaga jedynie, aby oznaczać testowane metody adnotacją `@Test`. Metody testowe mogą nie przyjmować argumentów i zwracać wartości `boolean` sygnalizujące powodzenie albo błąd. Metody testowe mogą mieć dowolne nazwy. Do tego metody testowe @Unit mogą odwoływać się do dowolnych pól klasy, również pól prywatnych.

Aby użyć infrastruktury @Unit, należy zaimportować do własnego programu pakiet `net.mindview.atunit`[8], oznaczyć wybrane pola i metody znacznikami @Unit (ich działanie poznasz przy okazji kolejnych przykładów), a potem zmusić system kompilacji do uruchomienia testów dla tak powstałej klasy. Oto prosty przykład:

```
//: annotations/AtUnitExample1.java
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample1 {
  public String methodOne() {
    return "Metoda methodOne";
  }
  public int methodTwo() {
    System.out.println("Metoda methodTwo");
    return 2;
  }
  @Test boolean methodOneTest() {
    return methodOne().equals("Metoda methodOne");
  }
  @Test boolean m2() { return methodTwo() == 2; }
  @Test private boolean m3() { return true; }
  // Demonstracja wyjścia dla testów nieudanych:
  @Test boolean failureTest() { return false; }
  @Test boolean anotherDisappointment() { return false; }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
      "java net.mindview.atunit.AtUnit AtUnitExample1");
  }
} /* Output:
annotations.AtUnitExample1
. methodOneTest
. m2 Metoda methodTwo

. m3
. failureTest (nieudany)
. anotherDisappointment (nieudany)
(5 test(y)(ów))

>>> 2 NIEUDANE(-NYCH) <<<
```

[8] Biblioteka ta wchodzi w skład pakietu kodu rozprowadzanego wraz z książką.

```
  annotations.AtUnitExample1: failureTest
  annotations.AtUnitExample1: anotherDisappointment
*///:~
```

Klasy testowane przez @Unit muszą być umieszczane w pakietach.

Adnotacja `@Test` poprzedzająca metody `methodOneTest()`, `m2()`, `m3()`, `failureTest()` i `anotherDisappointment()` nakazuje systemowi testowania @Unit uruchamianie tych metod jako jednostek testowych. Przy okazji obecność tego znacznika to sygnał, że metoda nie przyjmuje argumentów i zwraca wartość `boolean` albo nie zwraca wartości wcale (`void`). Programista piszący metodę testową musi jedynie określić warunki powodzenia testu i zwrócić `true` (w przypadku powodzenia) albo `false` (jeśli test się nie powiedzie).

Kto korzystał już z JUnit, zauważy zapewne, że @Unit daje więcej informacji — na wyjściu powyższego programu widać kolejno uruchamiane testy i ich wyniki, a na końcu wypisywane jest podsumowanie wykrytych błędów ze wskazaniem na testy, które je wykryły.

Jeśli osadzanie metod testowych w klasach z jakichś względów jest niepożądane, można z tego zrezygnować. Najprościej wyodrębnić wtedy testy do klasy pochodnej:

```
//: annotations/AtUnitExternalTest.java
// Testy poza testowanymi klasami.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExternalTest extends AtUnitExample1 {
  @Test boolean _methodOne() {
    return methodOne().equals("Metoda methodOne");
  }
  @Test boolean _methodTwo() { return methodTwo() == 2; }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
     "java net.mindview.atunit.AtUnit AtUnitExternalTest");
  }
} /* Output:
annotations.AtUnitExternalTest
. _methodOne
. _methodTwo Metoda methodTwo

OK (2 test(y)(ów))
*///:~
```

Powyższy przykład ilustruje też zaletę swobody doboru nazw (w porównaniu do wymogów JUnit zakładających obecność w nazwach metod testujących przedrostka `test`). Metody oznaczone adnotacją `@Test` są tu metodami testującymi wprost metody docelowe; ich nazwy są identyczne z nazwami metod testowanych, z dodatkiem znaku podkreślenia (co nie znaczy, że uważam taką konwencję nazewniczą za jedyną słuszną — chodzi właśnie o swobodę doboru nazw do potrzeb).

Wyodrębnianie testów poza klasę testowaną może się także odbywać na bazie kompozycji:

```
//: annotations/AtUnitComposition.java
// Testy poza testowanymi klasami.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitComposition {
  AtUnitExample1 testObject = new AtUnitExample1();
  @Test boolean _methodOne() {
    return
      testObject.methodOne().equals("Metoda methodOne");
  }
  @Test boolean _methodTwo() {
    return testObject.methodTwo() == 2;
  }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
     "java net.mindview.atunit.AtUnit AtUnitComposition");
  }
} /* Output:
annotations.AtUnitComposition
. _methodOne
. _methodTwo Metoda methodTwo

OK (2 test(y)(ów))
*///:~
```

Dla każdego testu tworzony jest nowy egzemplarz obiektu testowanego (bo każdy test to osobny obiekt `AtUnitComposition`).

Nie ma tu żadnych specjalnych metod asercji, jak w JUnit, ale druga postać adnotacji `@Test` pozwala metodom testującym zwracać wartości `void`. W celu sprawdzania wyniku testu można skorzystać z instrukcji `assert` z języka Java. Asercje Javy trzeba normalnie włączyć opcją `-ea` w wierszu wywołaniu programu `java`, ale @Unit robi to za nas. Do sygnalizowania niepowodzenia testu można zaangażować nawet wyjątki. Niespełniona asercja albo wyjątek zgłaszany przez metodę testującą są interpretowane jako niepowodzenie testu, ale @Unit w takim przypadku bynajmniej nie przerywa testowania — kontynuuje wykonanie wszystkich pozostałych testów. Oto przykład:

```
//: annotations/AtUnitExample2.java
// Asercje i wyjątki w metodach testujących.
package annotations;
import java.io.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample2 {
  public String methodOne() {
    return "Metoda methodOne";
  }
  public int methodTwo() {
    System.out.println("Metoda methodTwo");
    return 2;
  }
  @Test void assertExample() {
    assert methodOne().equals("Metoda methodOne");
```

```
  }
  @Test void assertFailureExample() {
    assert 1 == 2: "Co za niespodzianka!";
  }
  @Test void exceptionExample() throws IOException {
    new FileInputStream("nofile.txt"); // Zgłasza wyjątek
  }
  @Test boolean assertAndReturn() {
    // Assercja z komunikatem:
    assert methodTwo() == 2: "Metoda methodTwo powinna zwracać 2";
    return methodOne().equals("Metoda methodOne");
  }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
      "java net.mindview.atunit.AtUnit AtUnitExample2");
  }
} /* Output:
annotations.AtUnitExample2
 . assertExample
 . assertFailureExample java.lang.AssertionError: Co za niespodzianka!
(nieudany)
 . exceptionExample java.io.FileNotFoundException: nofile.txt (System nie może odnaleźć określonego
pliku)
(nieudany)
 . assertAndReturn Metoda methodTwo

(4 test(y)(ów))

>>> 2 NIEUDANE(-NYCH) <<<
 annotations.AtUnitExample2: assertFailureExample
 annotations.AtUnitExample2: exceptionExample
*///:~
```

A tu przykład użycia testów z asercjami wyodrębnionych poza klasę testowaną; testy (zresztą bardzo proste) dotyczą klasy `java.util.HashSet`:

```
//: annotations/HashSetTest.java
package annotations;
import java.util.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class HashSetTest {
  HashSet<String> testObject = new HashSet<String>();
  @Test void initialization() {
    assert testObject.isEmpty();
  }
  @Test void _contains() {
    testObject.add("jeden");
    assert testObject.contains("jeden");
  }
  @Test void _remove() {
    testObject.add("jeden");
    testObject.remove("jeden");
    assert testObject.isEmpty();
  }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
```

```
        "java net.mindview.atunit.AtUnit HashSetTest");
    }
} /* Output:
annotations.HashSetTest
. initialization
. _remove
. _contains
OK (3 test(y)(ów))
*///:~
```

W porównaniu kompozycji z dziedziczeniem to ostatnie wydaje się (przy braku innych ograniczeń) prostsze.

Ćwiczenie 4. Sprawdź, czy przed każdym testem faktycznie tworzony jest nowy egzemplarz `testObject` (3).

Ćwiczenie 5. Zmodyfikuj powyższy przykład tak, aby wykorzystywał dziedziczenie (1).

Ćwiczenie 6. Wzorując się na programie *HashSetTest.java*, przetestuj kontener `LinkedList` (1).

Ćwiczenie 7. Zmodyfikuj poprzednie ćwiczenie tak, aby klasa testująca dziedziczyła po klasie testowanej (1).

W każdym teście jednostkowym @Unit tworzy obiekt testowanej klasy za pomocą jej konstruktora domyślnego. Utworzony obiekt poddawany jest testowi, a potem zostaje odrzucony, co ma zapobiec wpływowi efektów ubocznych jednych testów na pozostałe. Jak się rzekło, tworzenie kolejnych egzemplarzy testowych odbywa się za pośrednictwem domyślnego konstruktora klasy. Jeśli klasa nie posiada konstruktora domyślnego, albo test wymaga większej kontroli nad procesem tworzenia obiektu testowego, można utworzyć statyczną metodę konstruującą obiekt i opatrzyć ją adnotacją `@TestObjectCreate`, jak tutaj:

```
//: annotations/AtUnitExample3.java
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample3 {
  private int n;
  public AtUnitExample3(int n) { this.n = n; }
  public int getN() { return n; }
  public String methodOne() {
    return "Metoda methodOne";
  }
  public int methodTwo() {
    System.out.println("Metoda methodTwo");
    return 2;
  }
  @TestObjectCreate static AtUnitExample3 create() {
    return new AtUnitExample3(47);
  }
  @Test boolean initialization() { return n == 47; }
  @Test boolean methodOneTest() {
    return methodOne().equals("Metoda methodOne");
```

```
  }
  @Test boolean m2() { return methodTwo() == 2; }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
      "java net.mindview.atunit.AtUnit AtUnitExample3");
  }
} /* Output:
annotations.AtUnitExample3
 . initialization
 . methodOneTest
 . m2 Metoda methodTwo

OK (3 test(y)(ów))
*///:~
```

Metoda `@TestObjectCreate` musi być metodą statyczną i musi zwracać obiekt testowanej klasy — program @Unit sprawdzi, czy ten warunek został spełniony.

Niekiedy obsługa testu jednostkowego wymaga utworzenia dodatkowych pól klasy. Do oznaczania pól wykorzystywanych jedynie w teście (tak, aby można je było automatycznie usunąć przed dostarczeniem gotowego programu klientowi) służy adnotacja `@TestProperty`. Poniższy przykład wczytuje wartości z ciągu `String` podzielonego na wyrazy wywołaniem `String.split()`. Wartości wejściowe są wykorzystywane do tworzenia obiektów testowych:

```
//: annotations/AtUnitExample4.java
package annotations;
import java.util.*;
import net.mindview.atunit.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class AtUnitExample4 {
  static String theory = "Wszystkie brontozaury " +
    "są szczupłe na końcach i ZNACZNIE grubsze w środku tułowia" +
    ". które zwęża się znów ku początkowi.";
  private String word;
  private Random rand = new Random(); // Inicjowanie generatora odczytem czasu
  public AtUnitExample4(String word) { this.word = word; }
  public String getWord() { return word; }
  public String scrambleWord() {
    List<Character> chars = new ArrayList<Character>();
    for(Character c : word.toCharArray())
      chars.add(c);
    Collections.shuffle(chars, rand);
    StringBuilder result = new StringBuilder();
    for(char ch : chars)
      result.append(ch);
    return result.toString();
  }
  @TestProperty static List<String> input =
    Arrays.asList(theory.split(" "));
  @TestProperty
    static Iterator<String> words = input.iterator();
  @TestObjectCreate static AtUnitExample4 create() {
    if(words.hasNext())
      return new AtUnitExample4(words.next());
```

```
    else
      return null;
  }
  @Test boolean words() {
    print("'" + getWord() + "'");
    return getWord().equals("Wszystkie");
  }
  @Test boolean scramble1() {
    // Ustalanie generatora pod kątem powtarzalności wyników:
    rand = new Random(47);
    print("'" + getWord() + "'");
    String scrambled = scrambleWord();
    print(scrambled);
    return scrambled.equals("oyarnzrbtou");
  }
  @Test boolean scramble2() {
    rand = new Random(74);
    print("'" + getWord() + "'");
    String scrambled = scrambleWord();
    print(scrambled);
    return scrambled.equals("są");
  }
  public static void main(String[] args) throws Exception {
    System.out.println("zaczynamy");
    OSExecute.command(
      "java net.mindview.atunit.AtUnit AtUnitExample4");
  }
} /* Output:
zaczynamy
annotations.AtUnitExample4
 . words 'Wszystkie'

 . scramble1 'brontozaury'
oyarnzrbtou

 . scramble2 'są'
są

OK (3 test(y)(ów))
*///:~
```

Adnotacja `@TestProperty` może też być wykorzystywana do oznaczania metod używanych wyłącznie w czasie testów (ale niebędących metodami testującymi).

Zauważ, że ten konkretny program uzależnia poprawność testów od kolejności ich wykonywania (przez wybieranie kolejnych wyrazów ciągu wejściowego), co generalnie nie jest pożądane.

Jeśli konstrukcja obiektu testowego obejmuje inicjalizację, której powinny potem towarzyszyć operacje porządkujące, można do klasy testującej dodać statyczną metodę ze znacznikiem `@TestObjectCleanup`, która zadba o przygotowanie obiektu do porzucenia. W poniższym przykładzie metoda opatrzona znacznikiem `@TestObjectCreate` dla każdego egzemplarza obiektu testowego otwiera plik, który wypadałoby zamknąć przed porzuceniem obiektu:

```
//: annotations/AtUnitExample5.java
package annotations;
import java.io.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample5 {
  private String text;
  public AtUnitExample5(String text) { this.text = text; }
  public String toString() { return text; }
  @TestProperty static PrintWriter output;
  @TestProperty static int counter;
  @TestObjectCreate static AtUnitExample5 create() {
    String id = Integer.toString(counter++);
    try {
      output = new PrintWriter("Test" + id + ".txt");
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    return new AtUnitExample5(id);
  }
  @TestObjectCleanup static void
  cleanup(AtUnitExample5 tobj) {
    System.out.println("Operacje porządkowe");
    output.close();
  }
  @Test boolean test1() {
    output.print("test1");
    return true;
  }
  @Test boolean test2() {
    output.print("test2");
    return true;
  }
  @Test boolean test3() {
    output.print("test3");
    return true;
  }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
      "java net.mindview.atunit.AtUnit AtUnitExample5");
  }
} /* Output:
annotations.AtUnitExample5
 . test1
Operacje porządkowe
 . test2
Operacje porządkowe
 . test3
Operacje porządkowe
OK (3 test(y)(ów))
*///:~
```

Na wyjściu programu widać, że po każdym teście następowało automatyczne wywołanie metody porządkującej.

@Unit a typy ogólne

Typy ogólne sprawiają pewien kłopot, bo testy trudno uogólnić. Testować trzeba konkretny parametr typowy albo zestaw takich parametrów. Rozwiązanie jest proste: wystarczy wyprowadzić klasę testującą z konkretnej wersji klasy ogólnej.

Oto prosta uogólniona implementacja stosu:

```
//: annotations/StackL.java
// Stos implementowany na bazie kontenera LinkedList.
package annotations;
import java.util.*;

public class StackL<T> {
  private LinkedList<T> list = new LinkedList<T>();
  public void push(T v) { list.addFirst(v); }
  public T top() { return list.getFirst(); }
  public T pop() { return list.removeFirst(); }
} ///:~
```

Aby przetestować wersję stosu przechowującą elementy typu `String`, należy wyprowadzić klasę testującą z typu `StackL<String>`:

```
//: annotations/StackLStringTest.java
// Zdatność @Unit do testowania typów uogólnionych.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class StackLStringTest extends StackL<String> {
  @Test void _push() {
    push("jeden");
    assert top().equals("jeden");
    push("dwa");
    assert top().equals("dwa");
  }
  @Test void _pop() {
    push("jeden");
    push("dwa");
    assert pop().equals("dwa");
    assert pop().equals("jeden");
  }
  @Test void _top() {
    push("A");
    push("B");
    assert top().equals("B");
    assert top().equals("B");
  }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
      "java net.mindview.atunit.AtUnit StackLStringTest");
  }
} /* Output:
annotations.StackLStringTest
. push
```

```
. _pop
. _top
OK (3 test(y)(ów))
*///:~
```

Potencjalną wadą takiego dziedziczenia jest utrata możliwości odwoływania się do prywatnych metod klasy w czasie testu. Jeśli taka możliwość jest niezbędna, można albo tę metodę oznaczyć jako chronioną (`protected`), albo dodać do klasy typu uogólnionego nieprywatną metodę ze znacznikiem `@TestProperty`, która wywoła metodę prywatną (metoda oznaczona adnotacją `@TestProperty` zostanie usunięta z kodu produkcyjnego przez narzędzie `AtUnitRemover`, prezentowane w dalszej części rozdziału).

Ćwiczenie 8. Utwórz klasę z metodą prywatną i dodaj do niej nieprywatną metodę opatrzoną adnotacją `@TestProperty`. Wywołaj tę metodę w teście (2).

Ćwiczenie 9. Zaproponuj proste testy @Unit dla kontenera `HashMap` (2).

Ćwiczenie 10. Wybierz przykład z innej części książki i poddaj go testom @Unit (2).

Brak „pakietów" testowych

Jedną z większych zalet infrastruktury testowania @Unit wobec narzędzia JUnit jest brak konieczności grupowania testów za pomocą „pakietów". Otóż w JUnit trzeba było jakoś powiadamiać narzędzie testujące o zakresie testów, co sprowadzało się do tworzenia „pakietów" testów przeznaczonych do łącznego uruchamiania w ramach testu.

Tymczasem w @Unit odbywa się poszukiwanie plików klas zawierających odpowiednie adnotacje, a potem uruchamianie metod oznaczonych adnotacją `@Test`. Przy tworzeniu infrastruktury @Unit postawiłem sobie za cel maksymalną przezroczystość infrastruktury, tak aby programiści mogli po prostu pisać potrzebne im metody testujące bez dodatkowego kodu i wiedzy wymaganych przy stosowaniu JUnit i wielu innych (skądinąd świetnych) systemów tego typu. Projektowanie testów to wyzwanie samo w sobie, również bez zewnętrznych komplikacji — dlatego w @Unit starałem się całość maksymalnie uprościć. Być może dzięki temu ktoś faktycznie pokusi się o napisanie testów (bo w obliczu komplikacji często się z tego rezygnuje).

Implementacja @Unit

Implementacja infrastruktury testowania z użyciem adnotacji wymaga przede wszystkim zdefiniowania wszystkich typów adnotacji. Są one prostymi znacznikami pozbawionymi pól. Definicja znacznika `@Test` była już podawana na początku rozdziału; oto reszta adnotacji @Unit:

```
//: net/mindview/atunit/TestObjectCreate.java
// Znacznik @Unit @TestObjectCreate.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCreate {} ///:~
```

```
//: net/mindview/atunit/TestObjectCleanup.java
// Znacznik @Unit @TestObjectCleanup.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCleanup {} ///:~

//: net/mindview/atunit/TestProperty.java
// Znacznik @Unit @TestProperty.
package net.mindview.atunit;
import java.lang.annotation.*;

// Jako właściwości testowe mogą występować zarówno pola, jak i metody:
@Target({ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface TestProperty {} ///:~
```

Wszystkie znaczniki mają zasięg podtrzymania określony jako RUNTIME, bo system @Unit realizuje testy na bazie skompilowanego kodu.

Do zaimplementowania systemu uruchamiającego testy wykorzystamy refleksję, pomocną w wyłuskiwaniu adnotacji. Program wykorzysta pozyskane informacje do podejmowania decyzji co do sposobu konstruowania obiektów testowych i uruchamiania testów. Dzięki adnotacjom całość jest zaskakująco prosta i zwarta:

```
//: net/mindview/atunit/AtUnit.java
// Infrastruktura testów modułów na bazie adnotacji.
// {RunByHand}
package net.mindview.atunit;
import java.lang.reflect.*;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class AtUnit implements ProcessFiles.Strategy {
  static Class<?> testClass;
  static List<String> failedTests= new ArrayList<String>();
  static long testsRun = 0;
  static long failures = 0;
  public static void main(String[] args) throws Exception {
    ClassLoader.getSystemClassLoader()
      .setDefaultAssertionStatus(true); // Włączenie asercji
    new ProcessFiles(new AtUnit(), "class").start(args);
    if(failures == 0)
      print("OK (" + testsRun + " test(y)(ów))");
    else {
      print("(" + testsRun + " test(y)(ów))");
      print("\n>>> " + failures + " NIEUDAN" +
        (failures > 1 ? "E(-YCH)" : "Y") + " <<<");
      for(String failed : failedTests)
        print("  " + failed);
    }
  }
  public void process(File cFile) {
```

```
try {
  String cName = ClassNameFinder.thisClass(
    BinaryFile.read(cFile));
  if(!cName.contains("."))
    return; // Ignorowanie klas spoza pakietów
  testClass = Class.forName(cName);
} catch(Exception e) {
  throw new RuntimeException(e);
}
TestMethods testMethods = new TestMethods();
Method creator = null;
Method cleanup = null;
for(Method m : testClass.getDeclaredMethods()) {
  testMethods.addIfTestMethod(m);
  if(creator == null)
    creator = checkForCreatorMethod(m);
  if(cleanup == null)
    cleanup = checkForCleanupMethod(m);
}
if(testMethods.size() > 0) {
  if(creator == null)
    try {
      if(!Modifier.isPublic(testClass
          .getDeclaredConstructor().getModifiers())) {
        print("Błąd: " + testClass +
          " konstruktor domyślny musi być publiczny");
        System.exit(1);
      }
    } catch(NoSuchMethodException e) {
      // Wystarczy syntetyzowany konstruktor domyślny
    }
  print(testClass.getName());
}
for(Method m : testMethods) {
  printnb("  . " + m.getName() + " ");
  try {
    Object testObject = createTestObject(creator);
    boolean success = false;
    try {
      if(m.getReturnType().equals(boolean.class))
        success = (Boolean)m.invoke(testObject);
      else {
        m.invoke(testObject);
        success = true; // Jeśli wszystkie asercje były spełnione
      }
    } catch(InvocationTargetException e) {
      // Właściwy wyjątek tkwi w e:
      print(e.getCause());
    }
    print(success ? "" : "(nieudany)");
    testsRun++;
    if(!success) {
      failures++;
      failedTests.add(testClass.getName() +
        ": " + m.getName());
    }
    if(cleanup != null)
      cleanup.invoke(testObject, testObject);
```

```
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
}
static class TestMethods extends ArrayList<Method> {
  void addIfTestMethod(Method m) {
    if(m.getAnnotation(Test.class) == null)
      return;
    if(!(m.getReturnType().equals(boolean.class) ||
        m.getReturnType().equals(void.class)))
      throw new RuntimeException("Metoda @Test " +
        "musi zwracać void albo boolean");
    m.setAccessible(true); // Dla prywatnych
    add(m);
  }
}
private static Method checkForCreatorMethod(Method m) {
  if(m.getAnnotation(TestObjectCreate.class) == null)
    return null;
  if(!m.getReturnType().equals(testClass))
    throw new RuntimeException("Metoda @TestObjectCreate " +
      "musi zwracać egzemplarz testowanej klasy.");
  if((m.getModifiers() &
      java.lang.reflect.Modifier.STATIC) < 1)
    throw new RuntimeException("Metoda @TestObjectCreate " +
      "musi być statyczna.");
  m.setAccessible(true);
  return m;
}
private static Method checkForCleanupMethod(Method m) {
  if(m.getAnnotation(TestObjectCleanup.class) == null)
    return null;
  if(!m.getReturnType().equals(void.class))
    throw new RuntimeException("Metoda @TestObjectCleanup " +
      "nie może zwracać wartości.");
  if((m.getModifiers() &
      java.lang.reflect.Modifier.STATIC) < 1)
    throw new RuntimeException("Metoda @TestObjectCleanup " +
      "musi być statyczna.");
  if(m.getParameterTypes().length == 0 ||
     m.getParameterTypes()[0] != testClass)
    throw new RuntimeException("Metoda @TestObjectCleanup " +
      "musi przyjmować argument testowanego typu.");
  m.setAccessible(true);
  return m;
}
private static Object createTestObject(Method creator) {
  if(creator != null) {
    try {
      return creator.invoke(testClass);
    } catch(Exception e) {
      throw new RuntimeException("Nie można uruchomić metody " +
        "@TestObject (kreatora).");
    }
  } else { // użycie konstruktora domyślnego:
    try {
      return testClass.newInstance();
```

```
        } catch(Exception e) {
          throw new RuntimeException("Nie można utworzyć " +
            "obiektu testowego. Spróbuj metody @TestObject.");
        }
      }
    }
  } ///:~
```

Program *AtUnit.java* wykorzystuje narzędzie `ProcessFiles` z pakietu `net.mindview.util`. Klasa `AtUnit` za pośrednictwem swojej metody `process()` implementuje strategię `ProcessFiles.Strategy`. Dzięki temu egzemplarz `AtUnit` można przekazać do konstruktora klasy `ProcessFiles`. Drugi argument konstruktora instruuje egzemplarz `ProcessFiles` odnośnie wyszukiwania plików z rozszerzeniem *.class*.

W przypadku braku argumentów wiersza wywołania program będzie przeglądał podkatalogi katalogu bieżącego. W wywołaniu programu można jednak podać wiele argumentów, zarówno w postaci nazw plików klas, jak i katalogów do przeszukania. Ponieważ @Unit automatycznie wyszukuje klasy i metody przystosowane do testowania, nie trzeba się uciekać do żadnych „pakietów"[9].

Jednym z problemów wymagających rozwiązania w programie *AtUnit.java* przy wyszukiwaniu plików klas jest to, że nazwa pliku klasy nie odzwierciedla właściwej nazwy klasy (kwalifikowanej nazwy, a więc z nazwą pakietu). Aby dobrać się do tej informacji, należy poddać analizie plik klasy, co nie jest proste, ale nie jest też niemożliwe[10]. Dlatego zaraz po znalezieniu i otwarciu pliku *.class* program wczytuje z niego dane i przekazuje je do metody `ClassNameFinder.thisClass()`. Wkraczamy tu w „inżynierię kodu bajtowego", bo zniżamy się do analizy zawartości skompilowanego pliku klasy:

```
//: net/mindview/atunit/ClassNameFinder.java
package net.mindview.atunit;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ClassNameFinder {
  public static String thisClass(byte[] classBytes) {
    Map<Integer,Integer> offsetTable =
      new HashMap<Integer,Integer>();
    Map<Integer,String> classNameTable =
      new HashMap<Integer,String>();
    try {
      DataInputStream data = new DataInputStream(
        new ByteArrayInputStream(classBytes));
      int magic = data.readInt(); // 0xcafebabe
      int minorVersion = data.readShort();
      int majorVersion = data.readShort();
      int constant_pool_count = data.readShort();
      int[] constant_pool = new int[constant_pool_count];
      for(int i = 1; i < constant_pool_count; i++) {
```

[9] Nie bardzo jest dla mnie jasne, dlaczego konstruktor klasy testowanej musi być publiczny, ale jeśli nie jest, wywołanie `newInstance()` najzwyczajniej zawiesza się (nie zgłaszając wyjątku).

[10] Spędziliśmy nad tym z Jeremym Meyerem prawie cały dzień.

```
        int tag = data.read();
        int tableSize;
        switch(tag) {
          case 1: // UTF
            int length = data.readShort();
            char[] bytes = new char[length];
            for(int k = 0; k < bytes.length; k++)
              bytes[k] = (char)data.read();
            String className = new String(bytes);
            classNameTable.put(i, className);
            break;
          case 5: // LONG
          case 6: // DOUBLE
            data.readLong(); // ominięcie 8 bajtów
            i++; // konieczny specjalny przeskok
            break;
          case 7: // CLASS
            int offset = data.readShort();
            offsetTable.put(i, offset);
            break;
          case 8: // STRING
            data.readShort(); // ominięcie 2 bajtów
            break;
          case 3:  // INTEGER
          case 4:  // FLOAT
          case 9:  // FIELD_REF
          case 10: // METHOD_REF
          case 11: // INTERFACE_METHOD_REF
          case 12: // NAME_AND_TYPE
            data.readInt(); // ominięcie 4 bajtów;
            break;
          default:
            throw new RuntimeException("Zły znacznik " + tag);
        }
      }
      short access_flags = data.readShort();
      int this_class = data.readShort();
      int super_class = data.readShort();
      return classNameTable.get(
        offsetTable.get(this_class)).replace('/', '.');
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  // Pokaz:
  public static void main(String[] args) throws Exception {
    if(args.length > 0) {
      for(String arg : args)
        print(thisClass(BinaryFile.read(new File(arg))));
    } else
      // Przejrzenie całej gałęzi podkatalogów:
      for(File klass : Directory.walk(".", ".*\\.class"))
        print(thisClass(BinaryFile.read(klass)));
  }
} ///:~
```

Nie zamierzam wyjaśniać wszystkich szczegółów; mówiąc w skrócie: wszystkie pliki klas są generowane w oparciu o pewien format, który starałem się zaprezentować za pośrednictwem w miarę czytelnych nazw pól skojarzonych z elementami danych wybieranych ze strumienia wejściowego `ByteArrayInputStream`. Rozmiar każdego takiego elementu można odczytać z rozmiaru operacji odczytu ze strumienia. Na przykład pierwsze 32 bity każdego pliku klasy to „magiczna liczba" `0xcafebabe`[11], a następne dwie liczby typu `short` to informacja o wersji. Pula stałych zawiera stałe zdefiniowane w programie i jako taka ma zmienny rozmiar; następna wartość `short` podaje ten rozmiar, co umożliwia przydział odpowiednio dużej tablicy. Każdy element puli stałych może mieć ustalony albo zmienny rozmiar, trzeba więc szukać znaczników rozpoczynających każdą taką stałą, aby zdecydować o sposobie jej przetworzenia — do tego służy instrukcja `switch`. Nie musimy podejmować właściwej analizy wszystkich danych z pliku klasy, chcemy jedynie dotrzeć do tych, które są nam potrzebne, więc znaczna ilość danych jest w czasie analizy najzwyczajniej pomijana. Niezbędne nam informacje o klasie przechowujemy w `classNameTable` i `offsetTable`. Po przepatrzeniu puli stałych odnajdziemy informację `this_class` reprezentującą przesunięcie w tablicy `offsetTable`, które daje z kolei indeks tabeli `classNameTable` zawierającej nazwę klasy.

Wracamy do metody `process()` w programie *AtUnit.java*: mamy już nazwę klasy i możemy sprawdzić, czy zawiera kropkę, która oznaczałaby zawieranie się klasy w pakiecie. Jeśli klasa pochodzi z pakietu, można ją załadować za pomocą standardowego modułu ładującego klasy uruchamianego wywołaniem `Class.forName()`. Teraz klasę można już analizować pod kątem adnotacji @Unit.

Szukamy tylko trzech adnotacji: `@Test`, opatrujących metody (zachowujemy je na liście `TestMethods`), i opcjonalnych adnotacji `@TestObjectCreate` oraz `@TestObjectCleanup`. Są one wykrywane za pośrednictwem wywołań widocznych w kodzie źródłowym.

Jeśli uda się znaleźć jakiekolwiek metody testowe (z adnotacją `@Test`), na wyjściu wypisywana jest nazwa klasy, tak aby osoba obserwująca przebieg testu wiedziała, co się dzieje. Potem wykonywane są testy dla danej klasy. Testy polegają na wypisaniu nazwy metody testującej, a następnie wywołaniu metody `createTestObject()`, która tworzy egzemplarz obiektu testowanej klasy za pośrednictwem konstruktora domyślnego albo metody statycznej oznaczonej adnotacją `@TestObjectCreate`. Po utworzeniu obiektu testowego dochodzi do wywołania na jego rzecz metody testowej. Jeśli metoda zwraca wartość typu `boolean`, wynik ów jest przechwytywany. Jeśli nie, zakładamy powodzenie testu w przypadku braku wyjątku (który powinien się pojawić w przypadku niespełnionej asercji albo jakiegokolwiek innego wyjątku). W przypadku zgłoszenia wyjątku wypisujemy informacje wyłuskane z obiektu wyjątku na wyjściu programu, aby obserwator testu miał możliwość określenia przyczyny błędu. Każde niepowodzenie testu powoduje zwiększenie licznika niepowodzeń i dodanie nazwy klasy i metody do listy `failedTest`, która ostatecznie jest wypisywana na wyjściu w ramach podsumowania testu.

Ćwiczenie 11. Dodaj do infrastruktury @Unit adnotację `@TestNote` oznaczającą notkę wyświetlaną podczas testu i odrzucaną w kompilacji docelowej (5).

[11] Znaczenie tej liczby ginie w bezliku legend, ale ponieważ Java to dzieło typowych maniaków komputerowych, można zaryzykować stwierdzenie, że ma to coś wspólnego z fantazjami o kobiecie w kafejce (w języku angielskim *java* to gatunek kawy, którą pija się w kawiarni — *cafe*; z kolei *babe* to dziewczyna, kociak — *przyp. tłum.*).

Usuwanie kodu testującego

Choć w wielu projektach pozostawienie kodu testowego w ostatecznej kompilacji produktu nie wpływa na niego ujemnie (zwłaszcza, jeśli metody testujące są prywatne, co jest jak najbardziej dopuszczalne), zwykle jednak wypadałoby się pozbyć kodu testowego, choćby po to, aby zminimalizować rozmiar pakietu oprogramowania i nie eksponować elementów nieistotnych dla klienta.

Usuwanie kodu testowego wymaga sporej dłubaniny, to jest inżynierii kodu bajtowego o uciążliwości zbyt wielkiej, aby opłacało się ją przeprowadzać ręcznie. Na szczęście do dyspozycji mamy bibliotekę Javassist[12], która znacznie ułatwia postulowaną inżynierię. Poniższy program przyjmuje za pośrednictwem pierwszego argumentu wywołania opcję `-r`; jej obecność powoduje usunięcie adnotacji `@Test`, a jej nieobecność — jedynie wypisanie wszystkich takich adnotacji. Również tu do przeglądania katalogów w poszukiwaniu plików klas wykorzystywana jest klasa `ProcessFiles`:

```
// : net/mindview/atunit/AtUnitRemover.java
// Wypisuje adnotacje infrastruktury testowej @Unit znalezione
// w skompilowanych plikach klas. Jeśli pierwszy argument to "-r",
// adnotacje @Unit są dodatkowo usuwane.
// {Args: ..}
// {Requires: javassist.bytecode.ClassFile;
// Wymaga instalacji biblioteki Javassist dostępnej pod adresem
// http://sourceforge.net/projects/jboss/ }
package net.mindview.atunit;
import javassist.*;
import javassist.expr.*;
import javassist.bytecode.*;
import javassist.bytecode.annotation.*;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class AtUnitRemover
implements ProcessFiles.Strategy {
  private static boolean remove = false;
  public static void main(String[] args) throws Exception {
    if(args.length > 0 && args[0].equals("-r")) {
      remove = true;
      String[] nargs = new String[args.length - 1];
      System.arraycopy(args, 1, nargs, 0, nargs.length);
      args = nargs;
    }
    new ProcessFiles(
      new AtUnitRemover(), "class").start(args);
  }
  public void process(File cFile) {
    boolean modified = false;
    try {
      String cName = ClassNameFinder.thisClass(
```

[12] Jej twórcy, którym jest dr Shigeru Chiba, należą się podziękowania i za samo dzieło, i za wydatną pomoc przy opracowaniu programu *AtUnitRemover.java*.

```
        BinaryFile.read(cFile));
      if(!cName.contains("."))
        return; // Ignorowanie klas spoza pakietów
      ClassPool cPool = ClassPool.getDefault();
      CtClass ctClass = cPool.get(cName);
      for(CtMethod method : ctClass.getDeclaredMethods()) {
        MethodInfo mi = method.getMethodInfo();
        AnnotationsAttribute attr = (AnnotationsAttribute)
          mi.getAttribute(AnnotationsAttribute.visibleTag);
        if(attr == null) continue;
        for(Annotation ann : attr.getAnnotations()) {
          if(ann.getTypeName()
              .startsWith("net.mindview.atunit")) {
            print(ctClass.getName() + " Metoda: "
              + mi.getName() + " " + ann);
            if(remove) {
              ctClass.removeMethod(method);
              modified = true;
            }
          }
        }
      }
      // ta wersja nie usuwa pól (zobacz omówienie).
      if(modified)
        ctClass.toBytecode(new DataOutputStream(
          new FileOutputStream(cFile)));
      ctClass.detach();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
} ///:~
```

`ClassPool` to rodzaj przeglądu wszystkich klas systemu, które podlegają modyfikacji w ramach programu. Gwarantuje ona spójność wszystkich przetworzonych klas. Każdą z klas `CtClass` należy wybrać z puli `ClassPool`, podobnie jak przy ładowaniu klasy za pośrednictwem wywołania `Class.forName()`.

Obiekt `CtClass` zawiera kod bajtowy skompilowanej klasy i pozwala na wygenerowanie informacji o klasie oraz manipulowanie jej kodem. W naszym przykładzie wywołujemy metodę `getDeclaredMethods()` (zupełnie jak w interfejsie refleksji Javy) i pozyskujemy z każdego obiektu `CtMethod` obiekt `MethodInfo`. Z tego możemy z kolei wyłuskać adnotacje. Metoda posiadająca adnotacje z pakietu `net.mindview.atunit` jest usuwana.

W przypadku modyfikacji pierwotny plik klasy zostaje zamazany nową zawartością klasy.

W czasie przygotowywania niniejszej książki Javassist wzbogacał się właśnie[13] o implementację funkcji usuwania i odkryliśmy, że usuwanie pól `@TestProperty` jest bardziej skomplikowane niż usuwanie metod. Chodzi o to, że pola takie mogą być skojarzone z blokami inicjalizacji statycznej i nie można ich usuwać ot tak. Dlatego powyższa wersja kodu usuwa jedynie metody testujące @Unit. Polecam jednak monitorowanie witryny

[13] Dr Shigeru Chiba na naszą prośbę uprzejmie uzupełnił interfejs o wywołanie `CtClass.removeMethod()`.

WWW[14] Javassist w poszukiwaniu informacji o aktualizacjach — wcześniej czy później pojawi się tam zapewne funkcja usuwania pól. Tymczasem warto zaznaczyć, że metoda testowania zewnętrznego, prezentowana w *AtUnitExternalTest.java*, pozwala na proste usunięcie całości kodu testowego przez zwyczajne usunięcie pliku klasy testowej.

Podsumowanie

Adnotacje są mile widzianym i oczekiwanym dodatkiem do Javy. Są one strukturalizowane i podlegają kontroli typów, a pozwalają na uzupełnianie kodu o metadane bez nadmiernego gmatwania i zaciemniania kodu. Dodatkowo można je wykorzystać do eliminowania znoju pisania deskryptorów klas i innych automatycznie generowanych danych. Zastąpienie znacznika Javadoc `@deprecated` adnotacją `@Deprecated` to tylko jeden z sygnałów przydatności adnotacji do opisywania informacji o klasach i ich przewagach nad zwyczajnymi komentarzami.

Sama Java (w wydaniu SE5) posiada jedynie garstkę predefiniowanych adnotacji. Oznacza to, że o ile nie znajdziesz zewnętrznych bibliotek, będziesz samodzielnie tworzył adnotacje i ich procesory. Możesz przy tym liczyć na wsparcie ze strony narzędzia apt, które bierze na siebie zadanie kompilowania plików wygenerowanych w trakcie przetwarzania adnotacji, co znacznie ułatwia proces kompilacji, ale póki co interfejs programistyczny `mirror` jest dość ubogi i nie pozwala na dużo więcej niż identyfikowanie poszczególnych elementów definicji klas Javy. Po lekturze rozdziału wiesz już, że do inżynierii kodu bajtowego skompilowanych plików klas najlepiej wykorzystać bibliotekę Javassist, choć w prostszych przypadkach można poradzić sobie z ręczną analizą plików klas.

Taki stan rzeczy nie będzie trwał wiecznie — należy się spodziewać kolejnych udogodnień, a także włączenia adnotacji do standardowych narzędzi niezależnych twórców interfejsów programistycznych, bibliotek i szkieletów aplikacji. O zbliżającej się zmianie, jaką obecność adnotacji wymusi na praktyce programowania w Javie, przekonuje choćby prezentacja przykładowego systemu testowania @Unit.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

[14] Informacje o Javassist można znaleźć na stronie domowej dr Shigeru Chiby (*http://www.csg.is.titech.ac.jp/~chiba/javassist/*) oraz na stronach projektu JBoss, do którego wcielono Javassist (*http://www.jboss.org/products/javassist*) — *przyp. tłum.*

Rozdział 21.

Współbieżność

Aż do tej pory zajmowaliśmy się wyłącznie programowaniem sekwencyjnym. Bo wszystko w programie dzieje się po kolei.

Programowaniem sekwencyjnym da się rozwiązać znaczną liczbę zadań programistycznych, ale charakterystyka niektórych problemów sprawia, że wygodne albo nawet konieczne staje się uruchamianie kilku części programu równolegle — tak aby te części wykonywały się współbieżnie albo faktycznie równolegle (jeśli mają do dyspozycji osobne procesory).

Programowanie współbieżne może znakomicie przyspieszyć wykonanie programu bądź ułatwić modelowanie niektórych problemów, a czasem da się osiągnąć jedno i drugie. Ale opanowanie teorii i technik programowania współbieżnego to nie lada wyzwanie i wymaga wyjścia ponad to, czego dowiedziałeś się dotychczas z tej książki — to temat dość już zaawansowany. Niniejszy rozdział będzie siłą rzeczy stanowił jedynie wprowadzenie, ale żaden programista nie może się uważać za posiadającego umiejętność programowania współbieżnego, jeśli nie opanuje materiału z tego rozdziału.

Jak się niebawem przekonasz, faktycznym problemem współbieżności jest wzajemne oddziaływanie wątków funkcjonujących równolegle. Oddziaływanie takie może być tak subtelne i okazjonalne, że usprawiedliwia stwierdzenie, że programowanie współbieżne to programowanie niby deterministyczne, ale w praktyce okazuje się niedeterministycznym. Można napisać programy współbieżne tak, aby — przy odpowiedniej staranności — działały poprawnie. W praktyce jednak znacznie częściej bywa tak, że programy tylko wydają się poprawne, ale w pewnych okolicznościach ochoczo zawodzą. Sęk w tym, że tych okoliczności można nigdy nie doczekać, albo mogą się pojawiać tak rzadko, że nie uda się ich wykryć w ramach testów — ba, być może nawet napisanie kodu testującego, prowokującego takie okoliczności, będzie zadaniem nie do wykonania! W efekcie o błędach dowiesz się najprędzej od niezadowolonych użytkowników. I choćby z tego powodu warto zapoznać się z teoretycznymi podstawami problemów współbieżności — ich ignorowanie to prosta droga do porażki.

Współbieżność jawi się teraz jako igranie z ogniem, ale jeśli pocieszy Cię to choć trochę, to powiem, że to nawet dobrze. Choć Java SE5 znacząco ulepszyła obsługę współbieżności, wciąż nie można liczyć na hamulec bezpieczeństwa w postaci weryfikacji kodu współbieżnego w czasie kompilacji, czy choćby wskazówek ze strony sprawdzanych wyjątków. Współbieżność zostawia programistę samego na pastwę losu i tylko staranność połączona z podejrzliwością może zabezpieczyć kod przez niepożądanymi efektami.

Niektórzy sądzą, że współbieżność to temat zbyt zaawansowany, aby pojawiał się w książce wprowadzającej do programowania w danym języku. Chcieliby potraktować ten temat osobno, jako rozłączny z nauką programowania, a tych kilka przypadków programowania współbieżnego zdarzających się w codziennej praktyce programistycznej chcieliby skodyfikować za pomocą prostych idiomów. Dlaczego więc ja chcę borykać się z tak skomplikowanym zagadnieniem, jeśli mógłbym je pominąć?

Gdyby to było takie proste... Niestety, to nie programista decyduje o momencie, w którym program staje się wielowątkowy. To, że nigdy nie uruchomiłeś wątku własnoręcznie, nie oznacza, że możesz unikać pisania kodu wątkowego. Weźmy za przykład systemy WWW, które stanowią matecznik programów w języku Java; otóż najbardziej podstawowa klasa biblioteki WWW, klasa serwletu, jest wielowątkowa — wielowątkowość to jej nieodłączna cecha, bo maszyny serwerów WWW często dysponują wieloma procesorami i zawsze robią z nich użytek. Mimo pozornej prostoty serwletu, właściwe jego wykorzystanie wymaga świadomości zagadnień współbieżności. To samo tyczy się zresztą programowania interfejsu użytkownika, o czym przekonasz się w rozdziale „Graficzne interfejsy użytkownika". Choć biblioteki Swing i SWT dysponują mechanizmami zabezpieczającymi przed wzajemnym niekorzystnym oddziaływaniem wątków, nieznajomość podstaw współbieżności mocno utrudnia programowanie z użyciem tych bibliotek.

Java to język wielowątkowy, a kwestie współbieżności są w nim obecne niezależnie od świadomości programisty. W efekcie w użyciu znajduje się wiele takich programów Javy, które albo działają zupełnym przypadkiem, albo wciąż tajemniczo zawodzą właśnie z powodu nierozpoznanych słabości ich implementacji współbieżności. Niekiedy taka kruchość programu jest jedynie uciążliwa, ale czasem prowadzi do utraty ważnych danych, a wobec całkowitej nieświadomości zagadnień współbieżności często szuka się przyczyn błędów wszędzie, tylko nie tam, gdzie one tkwią. Problemy tego rodzaju mogą się ujawnić albo nasilić po wdrożeniu programu na komputerze wieloprocesorowym. W każdym razie znajomość trudności współbieżności daje programiście świadomość tego, że pozornie działający program może w pewnych warunkach zachowywać się niepoprawnie.

Programowanie współbieżne można porównać do wkroczenia do całkowicie nowego świata i naukę całkowicie nowego języka programowania, a przynajmniej grupy zupełnie nowych pojęć językowych. Opanowanie zagadnień programowania współbieżnego jest tak samo trudne jak opanowanie polimorfizmu. Przy pewnym wysiłku można poznać podstawowy mechanizm, jednak dogłębne poznanie zagadnienia wymaga zazwyczaj wnikliwych studiów i dobrego zrozumienia. Celem niniejszego rozdziału jest dostarczenie Ci solidnej wiedzy o podstawach współbieżności, abyś mógł zrozumieć pojęcia i tworzyć poprawne programy wielowątkowe. Strzeż się jednak zbytniej pewności siebie i, jeśli musisz napisać coś naprawdę skomplikowanego, sięgnij po książkę poświęconą współbieżności.

Oblicza współbieżności

Główny powód pomieszania, jakie wprowadza współbieżność, tkwi w wielości problemów rozwiązywanych przy użyciu współbieżności i wielości implementacji współbieżności, jak też w braku jasnego rozdzielenia tych dwóch kwestii. W efekcie efektywne

stosowanie współbieżności jest uwarunkowane znajomością wszystkich kwestii i przypadków specjalnych rozpoznanych w programowaniu współbieżnym.

Problemy rozwiązywane przy użyciu współbieżności można określić ogólnie jako „prędkość" i „przejrzystość projektu".

Szybsze wykonanie

Kwestia prędkości, czy też szybkości wykonania, jawi się pozornie prosto: żeby program działał szybciej, wystarczy podzielić go na kawałki i każdy z nich uruchomić na osobnym procesorze. Współcześnie, kiedy prawo Moore'a (o podwajaniu wydajności procesorów w równych odstępach czasu) traci na znaczeniu (przynajmniej dla układów konwencjonalnej elektroniki krzemowej), wzrost wydajności osiąga się nie przez przyspieszanie układów, a przez ich zwielokrotnianie. Aby przyspieszyć własne programy, trzeba się nauczyć wykorzystywać obecność dodatkowych procesorów. To jedno oblicze współbieżności.

Dysponując komputerem wyposażonym w wiele procesorów, można rozdzielać wątki pomiędzy poszczególne procesory, co w ogromnym stopniu może poprawić przepustowość. Rozwiązanie to jest często stosowane w potężnych, wieloprocesorowych serwerach internetowych, na których specjalne programy są w stanie rozdzielać pomiędzy procesory obsługę wielu wątków, a każdy obsługuje jedno żądanie.

Ale okazuje się, że współbieżność może zwiększyć wydajność programów również wtedy, kiedy są one uruchamiane na *jednym* procesorze.

Brzmi to dziwnie i jeśli się zastanowić, to faktycznie narzuty związane z obsługą współbieżności *muszą* wydłużyć wykonanie programu współbieżnego uruchamianego na pojedynczym procesorze w porównaniu z czasem sekwencyjnego wykonania wszystkich jego zrównoleglonych części. Narzut ten to konieczność tzw. *przełączania kontekstu*, czyli przełączania procesora pomiędzy zadaniami. Czyżby więc taniej było uruchamiać na pojedynczym procesorze pojedyncze zadania i darować sobie przełączanie kontekstu?

O zasadności programowania współbieżnego w takich warunkach przesądza kwestia *blokowania*. Otóż jeśli jedno z zadań nie może kontynuować wykonania, bo uniemożliwiają to okoliczności pozostające poza kontrolą tego zadania (zazwyczaj chodzi o operacje wejścia-wyjścia), mówimy, że zadanie czy wątek zostały *zablokowane*. Brak współbieżności powoduje, że wykonanie programu jest przerywane aż do momentu zniesienia niekorzystnych okoliczności (albo zajścia okoliczności oczekiwanych). Gdyby program korzystał z możliwości współbieżności, inne zadania w ramach tego programu mogłyby kontynuować wykonywanie, posuwając tym samym naprzód wykonanie całego programu. Faktycznie, uruchamianie programów współbieżnych na pojedynczych procesorach nie ma sensu, o ile żadne z zadań nie może zostać zablokowane.

Bardzo typowym przykładem zwiększenia wydajności w wyniku zrównoleglenia wykonania nawet w systemie jednoprocesorowym jest *programowanie sterowane zdarzeniami*. W rzeczy samej, jednym z najbardziej przekonujących powodów wprowadzenia współbieżności jest chęć poprawienia szybkości reakcji interfejsu użytkownika. Wyobraź sobie program wykonujący jakieś czasochłonne operacje, który podczas ich realizacji przestaje odpowiadać na polecenia użytkownika i zdaje się na nic nie reagować. Nawet jeśli

w programie jest przycisk „zakończ", program najwyraźniej nie ma zamiaru sprawdzać jego stanu co kilka wierszy kodu źródłowego. Taki kod byłby co najmniej dziwaczny, zresztą model ten nie gwarantowałby reakcji interfejsu, bo programista mógłby najzwyczajniej zapomnieć o osadzeniu kontroli stanu przycisku w każdej operacji. Ale przy braku współbieżności jedynym sposobem zapewnienia reaktywności interfejsu użytkownika jest właśnie okresowa kontrola stanu wejść programu. Jeśli program wyodrębni do obsługi interfejsu osobny wątek wykonania, to mimo że ten wątek będzie przez większość czasu zawieszony w oczekiwaniu, wrażenie reaktywności będzie znacznie lepsze.

Program musi jednocześnie kontynuować obliczenia i przekazywać sterowanie z powrotem do interfejsu użytkownika, aby móc odpowiadać na wydawane polecenia. Ale konwencjonalna metoda nie jest w stanie kontynuować wykonywanych operacji i jednocześnie przekazać sterowania do innego miejsca programu. W rzeczywistości zadanie to wydaje się niemożliwe, gdyż wymagałoby podzielenia procesora na dwoje. A jednak współbieżność sprawia dokładnie takie wrażenie (a w przypadku systemów wieloprocesorowych to coś więcej niż tylko wrażenie).

Jednym ze sposobów implementowania współbieżności jest implementowanie jej na poziomie systemu operacyjnego za pomocą tak zwanych *procesów*. Proces jest wykonującym się programem z własną przestrzenią adresową. *Wielozadaniowy* system operacyjny jest w stanie uruchomić więcej niż jeden proces (program), poprzez okresowe przydzielanie mu cykli procesora — chociaż z zewnątrz wygląda to tak, jakby każdy działał samotnie. Procesy są atrakcyjne, bo system operacyjny zwykle zajmuje się izolowaniem jednych procesów od drugich, tak że nie mogą na siebie oddziaływać wprost, co znacznie ułatwia programowanie. Dla porównania, w systemach współbieżności takich jak w Javie dochodzi do współużytkowania zasobów, takich jak pamięć czy wejście-wyjście, więc zasadniczą trudnością programowania współbieżnego staje się koordynacja dostępu do tych zasobów pomiędzy zadaniami, tak aby nie dochodziło do kolizji.

Podam prosty przykład użycia procesów systemu operacyjnego. Otóż przy pisaniu książki regularnie wykonywałem wiele zapasowych kopii bieżącego stanu książki. Jedną kopię umieszczałem w katalogu lokalnym, jedną na karcie pamięci, jedną na dysku Zip i jedną na zdalnym serwerze FTP. Aby zautomatyzować proces wykonywania kopii, napisałem prosty program (w języku Python, ale to nie ma wiele do rzeczy), który pakował książkę do pliku z numerem wersji w nazwie i potem odpowiednio rozprowadzał kopie takiego pliku. Początkowo wykonywanie kopii zaprogramowałem sekwencyjnie, więc program czekał na rozpoczęcie następnego transferu do momentu zakończenia poprzedniego. Potem zdałem sobie sprawę z tego, że czas trwania każdej z operacji kopiowania zależy od szybkości nośnika i wydajności operacji wejścia-wyjścia. Ponieważ korzystałem z wielozadaniowego systemu operacyjnego, mogłem zainicjować wszystkie operacje kopiowania jako osobne procesy i uruchomić je równolegle, co znacznie przyspieszyło całą archiwizację. Kiedy jeden proces był blokowany w oczekiwaniu na zakończenie operacji wejścia-wyjścia, inny mógł kontynuować wykonywanie.

To doskonały przykład współbieżności. Każde zadanie wykonywało się jako proces we własnej przestrzeni adresowej, nie było więc możliwości wzajemnego niekorzystnego oddziaływania zadań. Co ważniejsze, zadania te nie musiały wymieniać ze sobą informacji, bo były od siebie całkowicie niezależne. O poprawność kopiowania dbał system operacyjny. W efekcie przyspieszenie archiwizacji obyło się bez jakiegokolwiek ryzyka, całkiem za darmo.

Niektórzy uważają[1], że to jedyne sensowne podejście do współbieżności. Niestety, procesy podlegają ograniczeniom i cechują się narzutami, przez co nie nadają się do rozwiązywania wszelkich problemów z dziedziny współbieżności.

Niektóre języki programowania są zaprojektowane pod kątem izolacji współbieżnych zadań. Mowa o tak zwanych *językach funkcyjnych*, w których wywołania funkcji nie prowokują efektów ubocznych (a więc nie mogą ingerować w działanie innych funkcji) i mogą być przez to wykonywane jako zupełnie niezależne zadania. Przykładem może być język *Erlang*, obejmujący bezpieczny mechanizm komunikacji pomiędzy zadaniami. Jeśli okaże się, że część programu musi wykorzystywać współbieżność i zmontowanie tej części przysparza poważnych problemów, warto rozważyć utworzenie tego fragmentu w dedykowanym języku programowania, jak *Erlang*.

W Javie przyjęto model tradycyjny, zakładający dobudowanie obsługi wątków na bazie języka zasadniczo sekwencyjnego[2]. Zamiast rozwidlać procesy za pomocą systemu operacyjnego, Java tworzy wątki w obrębie pojedynczego procesu reprezentowanego przez działający program. Jedną z zalet takiego podejścia jest uniezależnienie od systemu operacyjnego, co w przypadku Javy ma zasadnicze znaczenie. Na przykład wersje systemu operacyjnego Macintosh (sprzed wydania OS X), które stanowiły dość istotną platformę docelową dla pierwszych wersji Javy, nie obsługiwały wielozadaniowości. Gdyby nie własna obsługa wielowątkowości w Javie, żaden współbieżny program w tym języku nie dałby się uruchomić na komputerach Macintosh i podobnych platformach, co zniweczyłoby model „raz napisać, wszędzie uruchamiać"[3].

Ulepszanie projektu

Program wykorzystujący wiele zadań na komputerze jednoprocesorowym i tak będzie wykonywany tylko przez jeden procesor, więc teoretycznie dałoby się napisać go tak, żeby nie trzeba go było dzielić na wątki. Jednakże współbieżność wątków daje nie lada korzyści organizacyjne: może znacznie uprościć projekt programu. Niektóre typy problemów, na przykład z dziedziny symulacji, trudno wręcz rozwiązać bez wsparcia ze strony współbieżności.

Większość ludzi miała styczność z przynajmniej jedną formą symulacji, jaką jest gra komputerowa, ewentualnie komputerowo generowane animacje w filmach. Symulacje angażują wiele oddziałujących na siebie elementów, z których każdy „działa na własną rękę". Choć można zauważyć, że na jednoprocesorowym komputerze każdy element symulacyjny jest popychany naprzód przez ów jeden procesor, to z programistycznego punktu widzenia łatwiej udawać, że każdy element symulacji posiada własny procesor i działa jako niezależne zadanie.

---

[1] Mówi o tym choćby Eric Raymond, w swojej książce *The Art of UNIX Programming* (Addison-Wesley, 2004).

[2] Można by rzec, że wmontowywanie współbieżności w język sekwencyjny to podejście zgubne, ale zapraszam do snucia własnych wniosków.

[3] Model ten nie został nigdy do końca spełniony i już się tak głośno o nim nie mówi. Jak na ironię, przyczyny niemożności osiągnięcia warunku powszechnej możliwości uruchamiania programów w Javie niezależnie od platformy mogą mieć częściowo źródło właśnie w podsystemie wielowątkowości — który w Javie SE5 mógł zostać poprawiony.

Rozbudowana symulacja może obejmować bardzo dużą liczbę zadań, z których każde reprezentuje jakiś element symulacji — elementami będą nie tylko znane z gier elfy i wiedźmy, ale często również pojedyncze kamienie czy drzwi. Tymczasem systemy wielowątkowe często ograniczają liczbę tworzonych wątków; niekiedy limit wynosi kilkadziesiąt lub kilkaset wątków. Owo ograniczenie znajduje się poza kontrolą programu — może zależeć od platformy, albo (jak w przypadku języka Java) od implementacji maszyny wirtualnej. W Javie można ogólnie założyć, że mało kiedy będzie można pozwolić sobie na wydelegowanie jednego wątku dla każdego elementu rozległej symulacji.

Typowym rozwiązaniem problemu niedoboru wątków jest *wielowątkowość kooperacyjna*. Wielowątkowość w Javie to *wielowątkowość z wywłaszczaniem*, co oznacza, że pewien mechanizm szeregowania zadań przydziela wątkowi kwant czasu wykonania, po upływie którego wstrzymuje wątek i przełącza się do innego wątku, tak aby każdy z wątków doczekał się swojej porcji czasu procesora. W systemie z kooperacją każde z zadań winno dobrowolnie zrzekać się wykonania; wymaga to od programisty świadomego szpikowania kodu wątku instrukcjami zrzekania się. Zalety systemu kooperacyjnego są dwojakie: przełączanie kontekstu jest w nim z reguły obarczone znacznie mniejszym narzutem niż w systemie wywłaszczeniowym, a do tego teoretycznie nie ma ograniczenia co do liczby niezależnych zadań, które mogą działać współbieżnie. Przy wielkiej liczbie elementów symulacji może to być najlepsze rozwiązanie. Trzeba jednak zaznaczyć, że nie wszystkie systemy kooperacyjne obsługują rozprowadzanie zadań pomiędzy procesorami, co z kolei jest ich poważną wadą.

Współbieżność jest bardzo użytecznym modelem przy implementowaniu nowoczesnych systemów *wymiany komunikatów*, które angażują niekiedy wiele niezależnych komputerów rozproszonych w sieci komputerowej — bo współbieżność oddaje wtedy faktyczne zachowanie systemu. W takim przypadku wszystkie procesy systemu działają w faktycznej izolacji od siebie i nie mają nawet możliwości współużytkowania zasobów. Trzeba jednak synchronizować transfer informacji pomiędzy procesami, tak aby w całości systemu nie dochodziło do gubienia komunikatów albo ich przetrzymywania. Nawet jeśli w bezpośredniej przyszłości nie planujesz intensywnie korzystać z zalet współbieżności, powinieneś ją rozumieć choćby po to, aby lepiej radzić sobie właśnie z systemami opartymi na wymianie komunikatów, które zaczynają dominować we współczesnych systemach rozproszonych.

Współbieżność oznacza pewne koszty, również objawiające się zwiększoną złożonością, ale zwykle opłaca się je ponosić, a to z racji zalet w postaci uproszczenia projektów, równoważeniu wykorzystania zasobów i wygody użytkownika. Wątki pozwalają na tworzenie projektów zakładających rozluźnienie powiązania pomiędzy komponentami; na drugiej szali mamy zaś koszt polegający na konieczności jawnego naśladowania w kodzie zachowania i interakcji, naturalnych dla osobnych wątków.

Podstawy wielowątkowości

Programowanie współbieżne pozwala na podzielenie programu na oddzielne, wykonywane niezależnie zadania. W modelu wielowątkowości każde z tych niezależnych zadań (zwanych też podzadaniami) posiada własny *wątek wykonania*. Wątek to pojedynczy

sekwencyjny przepływ sterowania w obrębie procesu. Pojedynczy proces może zatem posiadać wiele jednocześnie wykonywanych zadań, ale programuje się go tak, jakby każde z tych zadań dysponowało własnym procesorem. Podziałem czasu procesora zajmuje się odpowiedni podsystem i nie trzeba się o to w ogóle troszczyć.

Model wątków jest pewnym programowym ułatwieniem mającym uprościć jednoczesną realizację kilku czynności przez ten sam program. Dzięki wątkom procesor może cyklicznie przydzielać każdemu z nich fragment swego czasu[4]. Każdy z wątków ma świadomość, że procesor „posiada" cały czas do swej dyspozycji, jednak w rzeczywistości czas procesora jest dzielony pomiędzy wszystkie istniejące wątki. Wyjątkiem jest sytuacja, gdy program jest wykonywany na komputerze wyposażonym w wiele procesorów; na szczęście jedną z ogromnych zalet wątków jest odseparowanie programu od warstwy niskopoziomowej, dzięki czemu nie musi on widzieć, czy jest realizowany na komputerze z jednym czy też z wieloma procesorami. A zatem wątki pozwalają na tworzenie programów skalowanych w niezauważalny sposób — jeśli program działa zbyt wolno, bardzo łatwo można go przyspieszyć, dodając do komputera kolejne procesory. Współbieżność oraz wielowątkowość są najbardziej rozsądnymi sposobami wykorzystania systemów wieloprocesorowych.

Definiowanie zadań

Wątek wykonuje zadanie, więc musimy jakoś opisać to zadanie. Służy do tego interfejs `Runnable`. Aby zdefiniować zadanie, należy po prostu zaimplementować interfejs `Runnable` i napisać metodę `run()` stanowiącą właściwy kod zadania.

Poniższe przykładowe zadanie `LiftOff` wypisuje przebieg odliczania na stanowisku startowym:

```
//: concurrency/LiftOff.java
// Demonstracja interfejsu Runnable.

public class LiftOff implements Runnable {
  protected int countDown = 10; // Wartość domyślna
  private static int taskCount = 0;
  private final int id = taskCount++;
  public LiftOff() {}
  public LiftOff(int countDown) {
    this.countDown = countDown;
  }
  public String status() {
    return "#" + id + "(" +
      (countDown > 0 ? countDown : "Start!") + "), ";
  }
  public void run() {
    while(countDown-- > 0) {
```

[4] To dotyczy systemów z podziałem czasu (jak system Windows). Z kolei w systemie Solaris stosowany jest model współbieżności FIFO: bieżący wątek działa tak długo, dopóki nie zostanie zablokowany w oczekiwaniu na zdarzenie (operację we-wy) albo nie dojdzie do wybudzenia wątku o wyższym priorytecie. Oznacza to, że inne wątki o tym samym priorytecie nie zostaną podjęte, dopóki bieżący wątek nie odda procesora.

```
        System.out.print(status());
        Thread.yield();
      }
    }
  } ///:~
```

Pole id służy do rozróżniania poszczególnych egzemplarzy zadania. Jest polem finalnym, ponieważ po inicjalizacji nie powinno ulegać zmianom.

Metoda run() zadania praktycznie zawsze zawiera pewien rodzaj pętli, która wykonuje się, dopóki wątek nie będzie nam już potrzebny, a zatem trzeba ustalić warunek przerwania pętli (albo po prostu wyjść z pętli instrukcją return). Często metoda ta jest implementowana jako pętla nieskończona, co oznacza, że o ile nic wystąpią jakieś zewnętrzne czynniki, które zmuszą ją do przerwania, to będzie trwać bezustannie (w dalszej części rozdziału zobaczysz, w jaki sposób można bezpiecznie przerywać realizację wątku).

Wywołanie statycznej metody Thread.yield() wewnątrz run() to sugestia dla planisty wątków (ang. *thread scheduler* — część podsystemu wielowątkowości Javy przełączająca procesor pomiędzy zadaniami): „zrobiłem swoje w tym cyklu, teraz można na chwilę przełączyć procesor do innych zadań". Wywołanie yield() jest nieobowiązkowe, ale dzięki jego zastosowaniu przykład daje na wyjściu ciekawsze wyniki: wyraźniej widać przeplatanie się wątków.

W następnym przykładzie metoda run() zadania nie posiada własnego wątku — jest wywoływana wprost z metody main() programu (która zresztą działa we własnym wątku — program składa się zawsze z przynajmniej jednego wątku, właśnie funkcji main()):

```
// : concurrency/MainThread.java

public class MainThread {
  public static void main(String[] args) {
    LiftOff launch = new LiftOff();
    launch.run();
  }
} /* Output:
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Start!),
*///:~
```

Jeśli klasa implementuje interfejs Runnable, musi posiadać metodę run(), ale to jeszcze nic specjalnego — samo posiadanie takiej metody nie implikuje żadnych cech wątków. Aby uzyskać zachowanie typowe dla wątków, trzeba jeszcze jawnie skojarzyć zadanie z wątkiem.

Klasa Thread

Tradycyjny sposób zamiany obiektu Runnable na działające zadanie polega na przekazaniu obiektu do konstruktora klasy Thread. Poniższy przykład ilustruje sposób utworzenia wątku zadania LiftOff:

```
// : concurrency/BasicThreads.java
// Najbardziej podstawowe zastosowanie klasy Thread.
```

```
public class BasicThreads {
  public static void main(String[] args) {
    Thread t = new Thread(new LiftOff());
    t.start();
    System.out.println("Oczekiwanie na start");
  }
} /* Output: (90% match)
Oczekiwanie na start
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Start!),
*///:~
```

Konstruktor klasy `Thread` wymaga przekazania jedynie obiektu zadania — implementacji `Runnable`. Wywołanie metody `start()` klasy `Thread` powoduje przeprowadzenie inicjalizacji nowego wątku, a następnie wywołanie w nim metody `run()` obiektu zadania.

Choć metoda `start()` wydaje się wywoływać metodę długotrwałą (na wyjściu programu pojawia się wtedy napis „Oczekiwanie na start"), zwraca sterowanie do wywołującego niemal natychmiast. Nastąpiło tym samym jakby wywołanie metody `LiftOff.run()` bez oczekiwania na jej zakończenie, bo metoda ta wykonuje się w osobnym wątku — zaś w `main()` można kontynuować wykonanie (nic dotyczy to rzecz jasna wyłącznie wątku funkcji `main()` — nowe wątki można uruchamiać z poziomu dowolnych działających wątków). Program wykonuje więc niejako równocześnie dwie metody: `main()` i `LiftOff.run()`.

Program można łatwo uzupełniać o kolejne wątki, wykonujące kolejne zadania. Poniżej widać zgodny chór uruchomionych wątków[5]:

```
//: concurrency/MoreBasicThreads.java
// Kolejne wątki.

public class MoreBasicThreads {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new Thread(new LiftOff()).start();
    System.out.println("Oczekiwanie na start");
  }
} /* Output: (Sample)
Oczekiwanie na start
#0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8), #3(8), #4(8), #0(7), #1(7), #2(7), #3(7), #4(7), #0(6),
#1(6), #2(6), #3(6), #4(6), #0(5), #1(5), #2(5), #3(5), #4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3),
#2(3), #3(3), #4(3), #0(2), #1(2), #2(2), #3(2), #4(2), #0(1), #1(1), #2(1), #3(1), #4(1), #0(Start!), #1(Start!),
#2(Start!), #3(Start!), #4(Start!),
*///:~
```

Na wyjściu programu widać, że poszczególne wątki wykonania przeplatały się ze sobą. Przeplatanie wątków wynika z działalności planisty wątków. W komputerach o większej liczbie procesorów planista, zamiast przeplatać wątki, będzie je dyskretnie rozprowadzał po dostępnych procesorach[6].

---

[5] W tym przypadku wszystkie wątki zadania `LiftOff` zostały utworzone w metodzie `main()`, ale gdyby wątki zadania `LiftOff` były tworzone z poziomu różnych wątków, mogłyby posiadać zdublowane identyfikatory. Wyjaśnię to w dalszej części rozdziału.

[6] Nie dotyczy to niektórych najwcześniejszych wersji Javy.

Wyniki uzyskiwane podczas kolejnych uruchomień programu mogą być różne, gdyż mechanizm planowania zarządzający wykonywaniem wątków nie jest deterministyczny. W rzeczywistości podczas wykonywania tego prostego programu w różnych wersjach Javy mogą wystąpić ogromne różnice. Na przykład w poprzedniej wersji JDK wykonywane wątki nie były zmieniane zbyt często. Mogło się zatem zdarzyć, że najpierw został zakończony pierwszy wątek, następnie drugi i tak dalej. Nie różniło się to zbytnio od wywoływania metody, która za jednym zamachem wykonuje wszystkie iteracje pętli. W zasadzie jedyną różnicą było wydłużenie czasu koniecznego do utworzenia i konfiguracji wszystkich wątków. W przypadku korzystania z JDK 1.4 uzyskano poprawę działania mechanizmu planowania wątków, gdyż, jak można sądzić, każdy wątek jest obsługiwany regularnie. Ogólnie rzecz biorąc, firma Sun nie podaje informacji na temat takich różnic w działaniu JDK, a zatem nie można polegać na spójnym systemie obsługi wątków. Najlepszym rozwiązaniem w przypadku tworzenia kodu wykorzystującego wątki jest zachowanie jak najdalej idącego konserwatyzmu.

Gdy metoda `main()` tworzy obiekty `Thread`, nie zapamiętuje żadnych referencji do nich. W przypadku zwyczajnych obiektów bez wątpienia stałyby się one łupem odśmiecacza pamięci, nie dotyczy to jednak obiektów `Thread`. Każdy z tych obiektów „rejestruje się", a zatem w rzeczywistości gdzieś jest przechowywane odwołanie do niego. Dzięki temu odśmiecacz pamięci nie może usunąć obiektu aż do momentu zakończenia realizacji metody `run()` i przerwania wątku. Na wyjściu programów przykładowych widać zresztą, że program nie kończy się przed zakończeniem wszystkich wątków.

Ćwiczenie 1. Zaimplementuj interfejs `Runnable`. W metodzie `run()` wypisz komunikat i wywołaj metodę `yield()`. Powtórz to trzykrotnie, a potem powróć z `run()`. Umieść w konstruktorze komunikat początku wykonania, a przy końcu zadania wypisz komunikat końcowy. Utwórz pewną liczbę obiektów zadań i uruchom je we własnych wątkach (2).

Ćwiczenie 2. Wzorując się na programie *generics/Fibonacci.java*, napisz zadanie, które wygeneruje sekwencję `n` wyrazów ciągu Fibonacciego, gdzie `n` będzie podawane do konstruktora zadania. Utwórz pewną liczbę obiektów zadania i uruchom je we własnych wątkach (2).

Wykonawcy

Wykonawcy (ang. *executors*) z biblioteki `java.util.concurrent` (tylko w SE5) to środki upraszczania programowania współbieżnego przez zarządzanie tworzeniem obiektów klasy `Thread`. Obiekt klasy `Executor` stanowi pośrednik pomiędzy klientem a wykonaniem zadania; klient, zamiast uruchamiać zadanie wprost, zdaje się na obiekt wykonawcy. Obiekty klasy `Executor` pozwalają na zarządzanie wykonaniem zadań asynchronicznych, eliminując konieczność jawnego zarządzania cyklem życia wątków. W Javie SE5 (i SE6) to zalecana metoda uruchamiania wątków.

Możemy więc w programie *MoreBasicThreads.java* wykorzystać w miejsce klasy `Thread` klasę `Executor`. Za specyfikację zadania odpowiada obiekt `LiftOff`; jak we wzorcu projektowym *Command* („polecenie") eksponuje on pojedynczą metodę uruchamiającą zadanie. Klasa `ExecutorService` wie z kolei, jak skonstruować kontekst niezbędny do uruchomienia implementacji `Runnable`. W poniższym przykładzie wykonawca `CachedThreadPool` utworzy jeden wątek dla każdego przekazanego zadania. Zauważ, że obiekt

ExecutorService tworzony jest za pośrednictwem statycznej metody klasy Executors, która określa rodzaj obiektu-wykonawcy:

```
//: concurrency/CachedThreadPool.java
import java.util.concurrent.*;

public class CachedThreadPool {
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new LiftOff());
    exec.shutdown();
  }
} /* Output: (Sample)
#0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8), #2(8), #3(8), #4(8), #0(6), #1(7), #2(7), #3(7), #4(7),
#0(5), #1(6), #2(6), #3(6), #4(6), #0(4), #1(5), #2(5), #3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2),
#1(3), #2(3), #3(3), #4(3), #0(1), #1(2), #2(2), #3(2), #4(2), #0(Liftoff!), #1(1), #2(1), #3(1), #4(1),
#1(Start!), #2(Start!), #3(Start!), #4(Start!),
*///:~
```

Bardzo często pojedynczy obiekt wykonawcy służy do tworzenia i zarządzania wszystkimi zadaniami w systemie.

Wywołanie metody shutdown() to blokada uruchamiania następnych wątków przez dany obiekt Executor. Bieżący wątek (w tym przypadku wątek metody main()) będzie kontynuował wykonanie wszystkich zadań zainicjowanych przed wywołaniem shutdown(). Program zakończy działanie zaraz po tym, jak zakończy się ostatnie zadanie wykonawcy.

Wykonawcę rodzaju CachedThreadPool można z łatwością zastąpić innym rodzajem wykonawcy, na przykład wykonawcą FixedThreadPool, który do wykonywania zadań wykorzystuje pulę wątków o ograniczonym rozmiarze:

```
//: concurrency/FixedThreadPool.java
import java.util.concurrent.*;

public class FixedThreadPool {
  public static void main(String[] args) {
    // Argument konstruktora ogranicza rozmiar puli wątków:
    ExecutorService exec = Executors.newFixedThreadPool(5);
    for(int i = 0; i < 5; i++)
      exec.execute(new LiftOff());
    exec.shutdown();
  }
} /* Output: (Sample)
#0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8), #2(8), #3(8), #4(8), #0(6), #1(7), #2(7), #3(7), #4(7),
#0(5), #1(6), #2(6), #3(6), #4(6), #0(4), #1(5), #2(5), #3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2),
#1(3), #2(3), #3(3), #4(3), #0(1), #1(2), #2(2), #3(2), #4(2), #0(Liftoff!), #1(1), #2(1), #3(1), #4(1),
#1(Start!), #2(Start!), #3(Start!), #4(Start!),
*///:~
```

Wykonawca FixedThreadPool przeprowadza kosztowną operację alokacji puli wątków jednorazowo, na początku działania, dysponując od tego momentu ustaloną liczbą wątków. W ten sposób oszczędzamy czas wykonania, który w przypadku puli dynamicznej jest poświęcany na tworzenie nowych wątków dla kolejnych zadań. Ponadto w systemie sterowanym zdarzeniami procedury obsługi zdarzeń wymagają, aby ich wątki były obsłużone możliwie szybko, najlepiej właśnie przez wyciągnięcie gotowego wątku z puli.

Do tego stosowanie puli wątków o stałym rozmiarze uniemożliwia wyczerpanie zasobów podsystemu wielowątkowości wskutek powiększania puli.

Zauważ, że w pulach wątków dowolnych rodzajów do uruchamiania kolejnych zadań wykorzystuje się w miarę możliwości już istniejące wątki.

Choć w ramach przykładów będziemy wykorzystywać pulę `CachedThreadPool`, w kodzie produkcyjnym rozważ też stosowanie puli stałej `FixedThreadPool`. Wykonawca z pulą dynamiczną (`CachedThreadPool`) będzie tworzył nowe wątki w miarę potrzeb, a wątki zwolnione w międzyczasie wykorzysta do obsługi kolejnych zadań — to pożądane zachowanie w okresie testowym. Jeśli jednak pula dynamiczna spowoduje problemy, powinieneś skorzystać z `FixedThreadPool`.

Wykonawca rodzaju `SingleThreadExecutor` to szczególny przypadek puli wątków, o liczbie wątków równej jeden[7]. Warto go wykorzystać do długotrwałego uruchamiania pojedynczego zadania w osobnym wątku — przykładem może być zadanie nasłuchujące w oczekiwaniu na połączenia sieciowe. Wykonawca taki przydaje się też jednak dla krótkich zadań, które mają zostać wykonane w osobnym wątku — na przykład zadania aktualizacji lokalnego czy zdalnego rejestru bądź dziennika; może to być także wątek zadania rozprowadzającego zdarzenia w systemie sterowanym zdarzeniami.

Jeśli do wykonawcy `SingleThreadExecutor` przekażesz więcej niż jedno zadanie, zadania zostaną uszeregowane w kolejce i wykonywane kolejno — wszystkie w tym samym wątku. Widać to w następnym przykładzie: każde zadanie jest wykonywane do końca, dopiero potem uruchamiane jest następne. Podsumowując, `SingleThreadExecutor` szereguje otrzymywane zadania, utrzymując własną (ukrytą) kolejkę zadań oczekujących na wykonanie:

```
// concurrency/SingleThreadExecutor.java
import java.util.concurrent.*;

public class SingleThreadExecutor {
  public static void main(String[] args) {
    ExecutorService exec =
      Executors.newSingleThreadExecutor();
    for(int i = 0; i < 5; i++)
      exec.execute(new LiftOff());
    exec.shutdown();
  }
} /* Output:
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Start!), #1(9), #1(8), #1(7), #1(6), #1(5),
#1(4), #1(3), #1(2), #1(1), #1(Start!), #2(9), #2(8), #2(7), #2(6), #2(5), #2(4), #2(3), #2(2), #2(1), #2(Start!),
#3(9), #3(8), #3(7), #3(6), #3(5), #3(4), #3(3), #3(2), #3(1), #3(Start!), #4(9), #4(8), #4(7), #4(6), #4(5),
#4(4), #4(3), #4(2), #4(1), #4(Start!),
*///:~
```

W ramach kolejnego przykładu założymy dostępność pewnej liczby wątków uruchamiających zadania odwołujące się do systemu plików. Zadaniami można zarządzać za pomocą wykonawcy, zyskując pewność, że nie dojdzie do przeplatania się odwołań do

[7] Daje też często ważną gwarancję współbieżności, której nie oferują pozostałe rodzaje wykonawców — niemożność współbieżnego uruchomienia dwóch zadań. Taka pewność zmienia wymagania co do blokowania (o czym powiemy sobie w dalszej części rozdziału).

systemu plików z różnych zadań. Nie trzeba więc będzie synchronizować dostępu do zasobu wspólnego (a co ważniejsze, nie dojdzie do uszkodzenia systemu plików). Niekiedy jednak lepszym rozwiązaniem jest podjęcie wyzwania i zadbanie o synchronizację dostępu; `SingleThreadExecutor` pozwala uniknąć tego kłopotu na przykład w fazie prototypowania projektu. Szeregowanie zadań eliminuje konieczność szeregowania obiektów.

Ćwiczenie 3. Powtórz ćwiczenie 1. z użyciem różnych wykonawców omawianych w tym punkcie (1).

Ćwiczenie 4. Powtórz ćwiczenie 2. z użyciem różnych wykonawców omawianych w tym punkcie (1).

Zwracanie wartości z zadań

Każda implementacja `Runnable` jest osobnym zadaniem, metodą uruchamianą w osobnym wątku, ale niezwracającą wartości do wywołującego. Jeśli zadanie ma generować jakieś wyniki, należy w miejsce interfejsu `Runnable` zastosować interfejs `Callable`. Interfejs ten (nowość w Javie SE5) to typ ogólny z parametrem typowym reprezentującym typ wartości zwracanej z metody `call()` (która zastępuje metodę `run()`). Zadanie implementujące interfejs `Callable` jest uruchamiane za pomocą wywołania `submit()` klasy `ExecutorService`. Oto prosty przykład:

```
// : concurrency/CallableDemo.java
import java.util.concurrent.*;
import java.util.*;

class TaskWithResult implements Callable<String> {
  private int id;
  public TaskWithResult(int id) {
    this.id = id;
  }
  public String call() {
    return "Wynik wywołania TaskWithResult " + id;
  }
}

public class CallableDemo {
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    ArrayList<Future<String>> results =
      new ArrayList<Future<String>>();
    for(int i = 0; i < 10; i++)
      results.add(exec.submit(new TaskWithResult(i)));
    for(Future<String> fs : results)
      try {
        // Wywołanie get() blokuje wywołującego:
        System.out.println(fs.get());
      } catch(InterruptedException e) {
        System.out.println(e);
        return;
      } catch(ExecutionException e) {
        System.out.println(e);
```

```
      } finally {
        exec.shutdown();
      }
    }
} /* Output:
Wynik wywołania TaskWithResult 0
Wynik wywołania TaskWithResult 1
Wynik wywołania TaskWithResult 2
Wynik wywołania TaskWithResult 3
Wynik wywołania TaskWithResult 4
Wynik wywołania TaskWithResult 5
Wynik wywołania TaskWithResult 6
Wynik wywołania TaskWithResult 7
Wynik wywołania TaskWithResult 8
Wynik wywołania TaskWithResult 9
*///:~
```

Metoda `submit()` zwraca obiekt klasy `Future`, sparametryzowany dla typu wartości zwracanej zgodnie z implementacją `Callable`. Obiekt `Future` można zapytać, czy reprezentuje wykonane zadanie (metoda `isDone()`). Jeśli zadanie zostało zakończone i wygenerowało wynik, można go pobrać wywołaniem metody `get()`. Można też wywołać metodę `get()` bez uprzedniego sprawdzania dostępności wyników metodą `isDone()`; wtedy jednak trzeba się liczyć z tym, że wywołanie `get()` zablokuje wywołującego aż do zakończenia zadania i pozyskania wartości zwracanej. Można też wreszcie wywołać metodę `get()` z limitem czasu oczekiwania na wynik.

Przeciążona metoda `Executors.callable()` przyjmuje obiekt `Runnable` i tworzy obiekt `Callable`. Obiekt klasy `ExecutorService` też posiada metody służące do uruchamiania obiektów `Callable`.

Ćwiczenie 5. Zmodyfikuj ćwiczenie 2. tak, aby uruchamiane zadanie było implementacją interfejsu `Callable` zliczającą wartości kolejnych wyrazów ciągu Fibonacciego. Utwórz kilka zadań i wypisz na wyjściu otrzymane wyniki (2).

Usypianie — wstrzymywanie wątku

Prostym sposobem kontrolowania zachowania wątku jest metoda `sleep()`, która zawiesza wykonanie wątku na określony czas. Jeśli wywołania metody `yield()` umieszczone w przykładzie z zadaniem `LiftOff` zastąpimy wywołaniami `sleep()`, uzyskamy następujące rezultaty:

```
//: concurrency/SleepingTask.java
// Wywoływanie sleep() w celu czasowego zawieszenia wykonania wątku.
import java.util.concurrent.*;

public class SleepingTask extends LiftOff {
  public void run() {
    try {
      while(countDown-- > 0) {
        System.out.print(status());
        // Dawniej:
        // Thread.sleep(100);
        // W Javie SE5/6:
        TimeUnit.MILLISECONDS.sleep(100);
      }
```

```
    } catch(InterruptedException e) {
      System.err.println("Przerwany!");
    }
  }
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new SleepingTask());
    exec.shutdown();
  }
} /* Output:
#0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8), #3(8), #4(8), #0(7), #1(7), #2(7), #3(7), #4(7), #0(6),
#1(6), #2(6), #3(6), #4(6), #0(5), #1(5), #2(5), #3(5), #4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3),
#2(3), #3(3), #4(3), #0(2), #1(2), #2(2), #3(2), #4(2), #0(1), #1(1), #2(1), #3(1), #4(1), #0(Start!), #1(Start!),
#2(Start!), #3(Start!), #4(Start!),
*///:~
```

Wywołanie metody `sleep()` może spowodować zgłoszenie wyjątku `InterruptedException`, który jest u nas przechwytywany wewnątrz metody `run()`. Ponieważ wyjątki nie są propagowane do wątku metody `main()`, trzeba je obsłużyć lokalnie, w obrębie danego wątku.

Java SE5 przewiduje bardziej jawną postać wywołania `sleep()` w ramach klasy `TimeUnit`. Pozwala ono na wygodne i czytelne określenie jednostek, w jakich liczony jest podawany czas uśpienia wątku. `TimeUnit` służy też do realizacji konwersji pomiędzy jednostkami — wykorzystamy to w jednym z kolejnych przykładów.

Na platformie testowej rozkład wątków był niezwykle równomierny — wątki były uruchamiane w kolejności od zerowego do czwartego i tak w kółko. To sensowne i oczekiwane zachowanie, skoro po każdej instrukcji wyjścia wątek jest usypiany, co pozwala planiście wątków na przełączenie się do innego wątku i wznowienie jego wykonania. Jednak taki sekwencyjny efekt jest wynikiem wewnętrznej implementacji mechanizmu wielowątkowości i nie można na nim polegać, bo implementacja ta może się zmieniać zależnie od systemu operacyjnego. Jeśli musimy kontrolować kolejność realizacji wątków, najlepszym rozwiązaniem jest po prostu z nich zrezygnować i stworzyć grupę współpracujących procedur, wzajemnie kontrolujących swoje działanie.

Ćwiczenie 6. Utwórz klasę zadania, które zawiesza swoje wykonanie na losowo dobierany czasu z zakresu od 1 do 10 sekund, a następnie wypisuje wylosowany czas i kończy działanie. Utwórz i uruchom pewną liczbę takich zadań (regulowaną argumentem wywołania programu) (2).

Priorytet wątku

Priorytet to dla planisty wątków informacja o poziomie ważności danego wątku. Co prawda kolejność, w jakiej procesor będzie wykonywać istniejącą grupę wątków, nie jest określona, ale jeśli na wznowienie oczekuje kilka zablokowanych wątków, to planista w pierwszej kolejności będzie wznawiał te, które mają najwyższy priorytet. Nie oznacza to wcale, że wątki o niższych priorytetach nie są w ogóle wykonywane (priorytety nigdy nie będą przyczyną zakleszczenia). Wątki o niższym priorytecie są po prostu wykonywane rzadziej.

W zdecydowanej większości przypadków wątki powinny mieć identyczne poziomy priorytetów. Ingerowanie w tę wartość rzadko kiedy jest konieczne, częściej jest wynikiem pomyłki.

Poniżej przedstawiłem przykład ilustrujący wpływ priorytetów na zachowanie wątków. Bieżący priorytet wątku można odczytać wywołaniem `getPriority()` i w dowolnym momencie zmienić go wywołaniem metody `setPriority()`.

```
//: concurrency/SimplePriorities.java
// Ilustracja zastosowania priorytetów wątków.
import java.util.concurrent.*;

public class SimplePriorities implements Runnable {
  private int countDown = 5;
  private volatile double d; // Bez optymalizacji
  private int priority;
  public SimplePriorities(int priority) {
    this.priority = priority;
  }
  public String toString() {
    return Thread.currentThread() + ": " + countDown;
  }
  public void run() {
    Thread.currentThread().setPriority(priority);
    while(true) {
      // Kosztowna czasowo operacja, dająca się przerywać:
      for(int i = 1; i < 100000; i++) {
        d += (Math.PI + Math.E) / (double)i;
        if(i % 1000 == 0)
          Thread.yield();
      }
      System.out.println(this);
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(
        new SimplePriorities(Thread.MIN_PRIORITY));
    exec.execute(
        new SimplePriorities(Thread.MAX_PRIORITY));
    exec.shutdown();
  }
} /* Output: (70% match)
Thread[pool-1-thread-6,10,main]: 5
Thread[pool-1-thread-6,10,main]: 4
Thread[pool-1-thread-6,10,main]: 3
Thread[pool-1-thread-6,10,main]: 2
Thread[pool-1-thread-6,10,main]: 1
Thread[pool-1-thread-3,1,main]: 5
Thread[pool-1-thread-2,1,main]: 5
Thread[pool-1-thread-1,1,main]: 5
Thread[pool-1-thread-5,1,main]: 5
Thread[pool-1-thread-4,1,main]: 5
...
*///:~
```

Metoda `toString()` została przesłonięta i wykorzystuje metodę `Thread.toString()`, prezentującą nazwę wątku, priorytet wątku oraz „grupę", do jakiej dany wątek należy (nazwę wątku można określić samodzielnie w konstruktorze; w naszym przypadku jest ona generowana automatycznie i ma postać: `pool-1-thread-1`, `pool-1-thread-2`, itd.). Przesłonięta wersja metody `toString()` wyświetla także licznik wątku. Zauważ, że wewnątrz zadania możesz pozyskać referencję obiektu wątku (`Thread`) wykonującego zadanie, wywołując metodę `Thread.currentThread()`.

Można zauważyć, że priorytet ostatniego wątku jest najwyższy oraz że wszystkie pozostałe wątki znajdują się na poziomach najniższych. Priorytet jest ustawiany na początku metody `run()`; ustawienie go w konstruktorze obiektu zadania byłoby nieskuteczne, bo w trakcie wykonania kodu konstruktora wykonawca nie uruchomił jeszcze zadania.

Wewnątrz metody `run()` wykonanych zostało 100 tysięcy powtórzeń raczej czasochłonnych operacji zmiennoprzecinkowych, polegających na dodawaniu i dzieleniu liczb `double`. Zmienna `d` została oznaczona jako `volatile`, by zapewnić, że korzystanie z niej nie będzie w żaden sposób optymalizowane. Bez tych operacji nie można by zauważyć efektu zmiany priorytetów (sprawdź sam — zakomentuj pętlę `for` zawierającą operacje zmiennoprzecinkowe). Dzięki realizowanym obliczeniom można się przekonać, że mechanizm zarządzający wątkami (przynajmniej na moim komputerze z systemem Windows XP) nadaje pierwszeństwo wątkowi z najwyższym priorytetem. Pomimo faktu, że wyświetlanie informacji na konsoli też jest operacją czasochłonną, bazując wyłącznie na niej, nie zauważymy efektu określania priorytetów, gdyż operacja wyjścia na konsolę nie daje się przerwać (w przeciwnym razie wyniki generowane przez programy wielowątkowe byłyby zniekształcone). Z kolei operacje matematyczne można przerywać. Wykorzystywane operacje matematyczne trwają na tyle długo, że mechanizm obsługi wątków może zadziałać i zmienić wykonywany wątek, uwzględniając przy tym priorytety, dzięki czemu wątek z najwyższym priorytetem uzyska pierwszeństwo. Aby dodatkowo się upewnić, że faktycznie dojdzie do przełączania kontekstów, w pętli wywołujemy regularnie metodę `yield()`.

Choć JDK udostępnia dziesięć poziomów priorytetów, nie we wszystkich systemach operacyjnych są dobrze odwzorowywane. Na przykład w systemie Windows siedem poziomów priorytetów nie zostało sztywno określonych, przez co ich odwzorowywanie nie jest deterministyczne (w systemie Solaris firmy Sun dostępnych jest 2^{31} poziomów priorytetów). Jedyną przenaszalną metodą modyfikacji priorytetów jest posługiwanie się stałymi `MAX_PRIORITY`, `NORM_PRIORITY` oraz `MIN_PRIORITY`.

Przełączanie

Gdy w danym przebiegu pętli metody `run()` zadania zrealizujesz już bieżącą cząstkę zadania, możesz przesłać mechanizmowi zarządzającemu wątkami podpowiedź, że aktualny wątek zrobił, co do niego należało, i można przydzielić procesor innemu wątkowi. Ta podpowiedź (faktycznie *jest* to jedynie podpowiedź, gdyż nie ma gwarancji, że implementacja mechanizmu zarządzania wątkami wykorzysta wskazówkę) przesyłana jest poprzez wywołanie metody `yield()`. Wywołanie `yield()` to sugestia możliwości uruchomienia innych wątków *o tym samym* priorytecie.

Przykład z odliczaniem (zadaniem `LiftOff`) wykorzystywał metodę `yield()` w celu osiągnięcia równomiernego rozkładu wykonania poszczególnych zadań odliczania. Spróbuj oznaczyć jako komentarz wywołanie `Thread.yield()` w metodzie `LiftOff.run()`, a (najprawdopodobniej) zobaczysz różnicę w zachowaniu zadań. Zasadniczo jednak nie powinieneś polegać na wywołaniu `yield()` w poważniejszym sterowaniu czy dostrajaniu przebiegu aplikacji. Metoda `yield()` jest więc często wykorzystywana niewłaściwie.

Wątki-demony

„Demon" to wątek, który powinien zapewnić ogólne usługi w tle programu w czasie jego działania, ale nie jest bezpośrednio związany z główną częścią programu. Kiedy więc wszystkie wątki nie będące demonami zakończą pracę, program również. Odwrotnie, jeżeli wciąż istnieją jakieś działające wątki nie będące demonami, to program się nie zakończy (istnieje na przykład niebędący demonem wątek, który uruchamia `main()`).

```
//: concurrency/SimpleDaemons.java
// Wątki-demony nie blokują zakończenia programu.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

public class SimpleDaemons implements Runnable {
  public void run() {
    try {
      while(true) {
        TimeUnit.MILLISECONDS.sleep(100);
        print(Thread.currentThread() + " " + this);
      }
    } catch(InterruptedException e) {
      print("przerwanie sleep()");
    }
  }
  public static void main(String[] args) throws Exception {
    for(int i = 0; i < 10; i++) {
      Thread daemon = new Thread(new SimpleDaemons());
      daemon.setDaemon(true); // Koniecznie wywołać przed metodą start()
      daemon.start();
    }
    print("Uruchomiono wszystkie demony");
    TimeUnit.MILLISECONDS.sleep(175);
  }
} /* Output: (Sample)
Uruchomiono wszystkie demony
Thread[Thread-0,5,main] SimpleDaemons@530daa
Thread[Thread-1,5,main] SimpleDaemons@a62fc3
Thread[Thread-2,5,main] SimpleDaemons@89ae9e
Thread[Thread-3,5,main] SimpleDaemons@1270b73
Thread[Thread-4,5,main] SimpleDaemons@60aeb0
Thread[Thread-5,5,main] SimpleDaemons@16caf43
Thread[Thread-6,5,main] SimpleDaemons@66848c
Thread[Thread-7,5,main] SimpleDaemons@8813f2
Thread[Thread-8,5,main] SimpleDaemons@1d58aae
Thread[Thread-9,5,main] SimpleDaemons@83cc67
...
*///:~
```

Należy określić wątek jako demona, zanim zostanie on uruchomiony. Do tego celu służy metoda `setDaemon()`.

Kiedy metoda `main()` zakończy swoje zadanie, nic nie powstrzymuje programu przed zakończeniem, ponieważ uruchomiliśmy tylko wątki-demony. Aby można było zaobserwować wynik uruchomienia wszystkich demonów, wątek `main()` jest na chwilę usypiany. Bez tego triku zobaczylibyśmy tylko parę wyników tworzenia wątków demonów (spróbuj zamienić długość uśpienia określaną w wywołaniu metody `sleep()`, aby zaobserwować działanie).

Program *SimpleDaemons.java* tworzy jawnie obiekty klasy `Thread`, aby mieć potem możliwość ustawiania znacznika „demoniczności". Ale atrybuty takie jak priorytet, demoniczność i nazwę można też dostosowywać w przypadku wątków tworzonych przez obiekty wykonawców — wystarczy napisać własną klasę wytwórni wątków:

```
//: net/mindview/util/DaemonThreadFactory.java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadFactory implements ThreadFactory {
  public Thread newThread(Runnable r) {
    Thread t = new Thread(r);
    t.setDaemon(true);
    return t;
  }
} ///:~
```

Od zwykłej wytwórni wątków `ThreadFactory` różni ją tylko to, że ustawia status demoniczności tworzonych wątków na `true`. Nową wytwórnię wątków możemy śmiało przekazać w wywołaniu metody `Executors.newCachedThreadPool()`:

```
//: concurrency/DaemonFromFactory.java
// Wykorzystanie wytwórni wątków-demonów.
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DaemonFromFactory implements Runnable {
  public void run() {
    try {
      while(true) {
        TimeUnit.MILLISECONDS.sleep(100);
        print(Thread.currentThread() + " " + this);
      }
    } catch(InterruptedException e) {
      print("Przerwano!");
    }
  }
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool(
      new DaemonThreadFactory());
    for(int i = 0; i < 10; i++)
      exec.execute(new DaemonFromFactory());
    print("Uruchomiono wszystkie demony");
    TimeUnit.MILLISECONDS.sleep(500); // Chwilowe uśpienie
  }
} /* (Execute to see output) *///:~
```

Każda ze statycznych metod kreacyjnych klasy ExecutorService jest przeciążona wersją przyjmującą obiekt wytwórni ThreadFactory, którą wykorzystuje do wytwarzania nowych wątków.

Możemy pójść o krok dalej i utworzyć pomocniczą klasę DaemonThreadPoolExecutor:

```
//: net/mindview/util/DaemonThreadPoolExecutor.java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadPoolExecutor
extends ThreadPoolExecutor {
  public DaemonThreadPoolExecutor() {
    super(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS,
      new SynchronousQueue<Runnable>(),
      new DaemonThreadFactory());
  }
} ///:~
```

Aby określić wartości odpowiednie dla konstruktora klasy bazowej, zajrzałem po prostu do pliku kodu źródłowego *Executors.java*.

Aby sprawdzić, czy wątek jest demonem, należy wywołać metodę isDaemon(). Jeśli wątek jest demonem, wtedy każdy wątek utworzony z jego wnętrza również będzie posiadał atrybut demoniczności, co ilustruje poniższy przykład:

```
//: concurrency/Daemons.java
// Wątki-demony wytwarzają kolejne demony.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Daemon implements Runnable {
  private Thread[] t = new Thread[10];
  public void run() {
    for(int i = 0; i < t.length; i++) {
      t[i] = new Thread(new DaemonSpawn());
      t[i].start();
      printnb("uruchomiono wątek DaemonSpawn " + i + ", ");
    }
    for(int i = 0; i < t.length; i++)
      printnb("t[" + i + "].isDaemon() = " +
        t[i].isDaemon() + ", ");
    while(true)
      Thread.yield();
  }
}

class DaemonSpawn implements Runnable {
  public void run() {
    while(true)
      Thread.yield();
  }
}

public class Daemons {
  public static void main(String[] args) throws Exception {
    Thread d = new Thread(new Daemon());
```

```
    d.setDaemon(true);
    d.start();
    printnb("d.isDaemon() = " + d.isDaemon() + ", ");
    // Umożliwiamy w wątkach-demonach zakończenie
    // ich procesów startowych:
    TimeUnit.SECONDS.sleep(1);
  }
} /* Output: (Sample)
d.isDaemon() = true, uruchomiono wątek DaemonSpawn 0, uruchomiono wątek DaemonSpawn 1,
uruchomiono wątek DaemonSpawn 2, uruchomiono wątek DaemonSpawn 3, uruchomiono wątek
DaemonSpawn 4, uruchomiono wątek DaemonSpawn 5, uruchomiono wątek DaemonSpawn 6,
uruchomiono wątek DaemonSpawn 7, uruchomiono wątek DaemonSpawn 8, uruchomiono wątek
DaemonSpawn 9, t[0].isDaemon() = true, t[1].isDaemon() = true, t[2].isDaemon() = true, t[3].isDaemon()
= true, t[4].isDaemon() = true, t[5].isDaemon() = true, t[6].isDaemon() = true, t[7].isDaemon() = true,
t[8].isDaemon() = true, t[9].isDaemon() = true,
*///:~
```

Wątek `Daemon` otrzymuje atrybut demoniczności, a następnie uruchamia pewną ilość kolejnych wątków, które bynajmniej *nie są* jawnie ustawiane jako demony, a które mimo to są demonami. Wątek `Daemon` oddaje sterowanie do pozostałych procesów, rozpoczynając pętlę nieskończoną z wywołaniami metody `yield()`.

Warto mieć świadomość, że wątki-demony będą kończyć wykonanie ich metod `run()` bez uruchamiania ewentualnych klauzul `finally`:

```
//: concurrency/DaemonsDontRunFinally.java
// Wątki-demony nie uruchamiają kodu z klauzuli finally metody run()
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class ADaemon implements Runnable {
  public void run() {
    try {
      print("Uruchomienie demona ADaemon");
      TimeUnit.SECONDS.sleep(1);
    } catch(InterruptedException e) {
      print("Wyjście przez wyjątek InterruptedException");
    } finally {
      print("Czy to się uruchomi?");
    }
  }
}

public class DaemonsDontRunFinally {
  public static void main(String[] args) throws Exception {
    Thread t = new Thread(new ADaemon());
    t.setDaemon(true);
    t.start();
  }
} /* Output:
Uruchomienie demona ADaemon
*///:~
```

Uruchomienie powyższego programu uwidoczni Ci fakt, że kod z klauzuli `finally` metody `run()` nie jest w ogóle wykonywany; zostałby natomiast wykonany, gdybyśmy oznaczyli jako komentarz wywołanie `setDaemon()`, czyniące wątek wątkiem-demonem.

To jak najbardziej poprawne zachowanie, mimo że nasze dotychczasowe doświadczenia z `finally` zdają się temu przeczyć. Otóż wątki-demony są przerywane gwałtownie, w momencie zakończenia wykonania ostatniego niedemonicznego wątku programu. Dlatego zaraz po zakończeniu metody `main()` maszyna wirtualna przerywa wykonywanie wszystkich demonów, nie dbając o formalności właściwe zwykłemu trybowi zakańczania procesów. Niemożność łagodnego i kontrolowanego przerwania wykonywania wątku-demona stanowi powód, aby używać ich wstrzemięźliwie. Zwykle lepiej sprawdzają się niedemoniczne obiekty-wykonawcy, bo pozwalają na zamykanie wątków pozostających pod kontrolą wykonawcy (wraz z możliwością zamknięcia wszystkich wątków danego wykonawcy za jednym zamachem). Jak się niebawem przekonasz, zamykanie odbywa się wtedy zgodnie z regułami sztuki.

Ćwiczenie 7. Poeksperymentuj z różnymi czasami uśpienia wątku w programie *Daemons.java* i zaobserwuj efekty (2).

Ćwiczenie 8. Zmodyfikuj program *MoreBasicThreads.java* tak, aby wszystkie występujące tam wątki były demonami; sprawdź, czy faktycznie cały program kończy się w momencie zakończenia metody `main()` (1).

Ćwiczenie 9. Zmodyfikuj program *SimplePriorities.java* tak, aby ustawiać priorytety wątków za pośrednictwem własnej wytwórni wątków `ThreadFactory` (3).

Wariacje na temat wątków

W prezentowanych dotąd przykładach klasy zadań implementowały interfejs `Runnable`. Tymczasem w najprostszych przypadkach można skutecznie definiować zadania jako bezpośrednie pochodne klasy `Thread`, jak tu:

```
//: concurrency/SimpleThread.java
// Dziedziczenie wprost po klasie Thread.

public class SimpleThread extends Thread {
  private int countDown = 5;
  private static int threadCount = 0;
  public SimpleThread() {
    // Ustawienie nazwy wątku:
    super(Integer.toString(++threadCount));
    start();
  }
  public String toString() {
    return "#" + getName() + "(" + countDown + "). ";
  }
  public void run() {
    while(true) {
      System.out.print(this);
      if(--countDown == 0)
        return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new SimpleThread();
  }
```

```
} /* Output:
#1(5), #1(4), #1(3), #1(2), #1(1), #2(5), #2(4), #2(3), #2(2), #2(1), #3(5), #3(4), #3(3), #3(2), #3(1), #4(5),
#4(4), #4(3), #4(2), #4(1), #5(5), #5(4), #5(3), #5(2), #5(1),
*///:~
```

Obiektom Thread nadaje się nazwy za pośrednictwem wywołania odpowiedniego konstruktora Thread. Tak ustawiona nazwa jest potem używana w metodzie toString() za pośrednictwem wywołania getName().

Można też spotkać twór w postaci samozarządzającego się obiektu Runnable:

```
//: concurrency/SelfManaged.java
// Implementacja Runnable z własnym wątkiem wykonania.

public class SelfManaged implements Runnable {
  private int countDown = 5;
  private Thread t = new Thread(this);
  public SelfManaged() { t.start(); }
  public String toString() {
    return Thread.currentThread().getName() +
      "(" + countDown + "), ";
  }
  public void run() {
    while(true) {
      System.out.print(this);
      if(--countDown == 0)
        return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new SelfManaged();
  }
} /* Output:
Thread-0(5), Thread-0(4), Thread-0(3), Thread-0(2), Thread-0(1), Thread-1(5), Thread-1(4), Thread-1(3),
Thread-1(2), Thread-1(1), Thread-2(5), Thread-2(4), Thread-2(3), Thread-2(2), Thread-2(1), Thread-3(5),
Thread-3(4), Thread-3(3), Thread-3(2), Thread-3(1), Thread-4(5), Thread-4(4), Thread-4(3), Thread-4(2),
Thread-4(1),
*///:~
```

Nie różni się to specjalnie od dziedziczenia po Thread, jedynie składnia jest nieco udziwniona, jednakże implementacja interfejsu pozwala na dziedziczenie po innych klasach, podczas gdy dziedziczenie po Thread nie daje tej możliwości.

Zwróć uwagę na wywołanie metody start() obecne w konstruktorze. Ten przykład jest bardzo uproszczony i pewnie dość bezpieczny, ale samo uruchamianie wątków w konstruktorach może być problematyczne, bo przed zakończeniem konstruktora może dojść do podjęcia innego zadania, co oznacza potencjalną możliwość odwołania się do obiektu pozostającego w stanie niestabilnym. To jeszcze jeden powód, aby tworzenie obiektów Thread złożyć na barki odpowiednich wykonawców.

Niekiedy warto ukryć kod wątkowy w klasie przez zastosowanie klasy wewnętrznej, jak tu:

```
//: concurrency/ThreadVariations.java
// Tworzenie wątków w klasach wewnętrznych.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;
```

```
// Zastosowanie nazwanej klasy wewnętrznej:
class InnerThread1 {
  private int countDown = 5;
  private Inner inner;
  private class Inner extends Thread {
    Inner(String name) {
      super(name);
      start();
    }
    public void run() {
      try {
        while(true) {
          print(this);
          if(--countDown == 0) return;
          sleep(10);
        }
      } catch(InterruptedException e) {
        print("przerwano");
      }
    }
    public String toString() {
      return getName() + ": " + countDown;
    }
  }
  public InnerThread1(String name) {
    inner = new Inner(name);
  }
}

// Zastosowanie anonimowej klasy wewnętrznej:
class InnerThread2 {
  private int countDown = 5;
  private Thread t;
  public InnerThread2(String name) {
    t = new Thread(name) {
      public void run() {
        try {
          while(true) {
            print(this);
            if(--countDown == 0) return;
            sleep(10);
          }
        } catch(InterruptedException e) {
          print("przerwano sleep()");
        }
      }
      public String toString() {
        return getName() + ": " + countDown;
      }
    };
    t.start();
  }
}

// Zastosowanie nazwanej wewnętrznej implementacji Runnable:
class InnerRunnable1 {
  private int countDown = 5;
  private Inner inner;
```

```
  private class Inner implements Runnable {
    Thread t;
    Inner(String name) {
      t = new Thread(this, name);
      t.start();
    }
    public void run() {
      try {
        while(true) {
          print(this);
          if(--countDown == 0) return;
          TimeUnit.MILLISECONDS.sleep(10);
        }
      } catch(InterruptedException e) {
        print("przerwano sleep()");
      }
    }
    public String toString() {
      return t.getName() + ": " + countDown;
    }
  }
  public InnerRunnable1(String name) {
    inner = new Inner(name);
  }
}

// Zastosowanie anonimowej wewnętrznej implementacji Runnable:
class InnerRunnable2 {
  private int countDown = 5;
  private Thread t;
  public InnerRunnable2(String name) {
    t = new Thread(new Runnable() {
      public void run() {
        try {
          while(true) {
            print(this);
            if(--countDown == 0) return;
            TimeUnit.MILLISECONDS.sleep(10);
          }
        } catch(InterruptedException e) {
          print("przerwano sleep()");
        }
      }
      public String toString() {
        return Thread.currentThread().getName() +
          ": " + countDown;
      }
    }, name);
    t.start();
  }
}

// Osobna metoda uruchamiająca pewien kod jako wątek:
class ThreadMethod {
  private int countDown = 5;
  private Thread t;
  private String name;
  public ThreadMethod(String name) { this.name = name; }
```

```
  public void runTask() {
    if(t == null) {
      t = new Thread(name) {
        public void run() {
          try {
            while(true) {
              print(this);
              if(--countDown == 0) return;
              sleep(10);
            }
          } catch(InterruptedException e) {
            print("przerwano sleep()");
          }
        }
        public String toString() {
          return getName() + ": " + countDown;
        }
      };
      t.start();
    }
  }
}

public class ThreadVariations {
  public static void main(String[] args) {
    new InnerThread1("InnerThread1");
    new InnerThread2("InnerThread2");
    new InnerRunnable1("InnerRunnable1");
    new InnerRunnable2("InnerRunnable2");
    new ThreadMethod("ThreadMethod").runTask();
  }
} /* (Execute to see output) *///:~
```

`InnerThread1` tworzy nazwaną klasę wewnętrzną rozszerzającą `Thread` i w konstruktorze tworzy instancję tej klasy wewnętrznej. Rozwiązanie to ma sens, jeśli klasa wewnętrzna dysponuje jakimiś szczególnymi możliwościami (nowymi metodami), które muszą być wykorzystywane w innych metodach. Jednak w większości przypadków głównym powodem przemawiającym za tworzeniem wątku jest wyłącznie chęć wykorzystania możliwości klasy `Thread`, a zatem nie ma potrzeby tworzenia nazwanej klasy wewnętrznej. Alternatywne rozwiązanie przedstawia klasa `InnerThread2`: W konstruktorze tworzona jest anonimowa klasa wewnętrzna dziedzicząca z `Thread`, która następnie, poprzez rzutowanie w górę, zwraca referencję `t` klasy `Thread`. Jeśli inne metody klasy wymagają dostępu do `t`, mogą go uzyskać za pośrednictwem interfejsu klasy `Thread`, dzięki czemu nie muszą znać dokładnego typu obiektu.

Pozostałe dwie klasy przedstawione w przykładzie stanowią powtórzenie dwóch poprzednich, przy czym wykorzystują interfejs `Runnable`, a nie klasę `Thread`.

Klasa `ThreadMethod` przedstawia sposób tworzenia wątku wewnątrz metody. Metoda jest wywoływana w momencie, gdy jesteśmy gotowi do uruchomienia wątku, a kończy się po jego uruchomieniu. Jeśli wątek wykonuje czynności pomocnicze i nie jest niezbędny do działania klasy, to prawdopodobnie jest to znacznie bardziej przydatny i właściwy sposób wykorzystania wątku niż uruchamianie go w konstruktorze klasy.

Ćwiczenie 10. Zmień ćwiczenie 5., wzorując się na klasie `ThreadMethod`, tak aby metoda `runTask()` przyjmowała argument w postaci liczby wyrazów ciągu Fibonacciego do zsumowania; każde wywołanie `runTask()` powinno zwracać obiekt `Future` wygenerowany za pomocą metody `submit()` (4).

Terminologia

Zaprezentowane omówienie dowodzi, że programy współbieżne można implementować na kilka sposobów, a ich wybór wcale nie musi być prosty. Często kłopot wynika z terminologii wykorzystywanej do opisu technologii współbieżnego wykonania programów, zwłaszcza tam, gdzie w grę wchodzą wątki.

Powinieneś już rozróżniać zadanie od wątku, w ramach którego to zadanie jest wykonywane; to rozróżnienie jest szczególnie wyraźne w bibliotekach języka Java, bo nad klasą wątku `Thread` nie mamy praktycznie żadnej kontroli (poziom izolacji jest jeszcze zwiększany przez obecność wykonawców, które przejmują zadanie tworzenia wątków i zarządzania nimi). Po prostu tworzymy wątek i kojarzymy go z zadaniem, tak że zadanie wykonuje się w danym wątku.

W języku Java klasa `Thread` sama z siebie nie realizuje żadnego zadania. Jej rolą jest wykonywanie zadania powierzonego. W literaturze poświęconej wielowątkowości można zwykle przeczytać, że „wątek robi to czy tamto". Powstaje wtedy wrażenie, że wątek jest tożsamy z zadaniem; sam zresztą, kiedy stykałem się po raz pierwszy z wątkami w Javie, doświadczyłem tego wrażenia tak silnie, że widziałem zadania jako twory „będące" wątkami, a więc zakładałem, że zadania powinny dziedziczyć po wątkach. Do tego nazwa interfejsu `Runnable` jest moim zdaniem nie najszczęśliwsza — interfejs ten powinien nosić miano bardziej kojarzące się z zadaniem (choćby `Task`). Gdyby interfejs stanowił jedynie zgrupowanie metod, wtedy przyjęte nazewnictwo „coś-tam-able" (sugerujące zdolność implementacji do „czegoś-tam") byłoby uzasadnione, ale skoro interfejs wyraża jednak pojęcie wyższego poziomu, jak „zadanie", nazwę `Runnable` uważam za nietrafioną.

Sęk w tym, że poziomy abstrakcji są ze sobą wymieszane. Na poziomie koncepcyjnym zamierzamy tworzyć zadania realizowane niezależnie od innych zadań; chcemy więc mieć możliwość zdefiniowania zadania i podjęcia jego realizacji („ruszaj!"), nie przejmując się detalami. Tymczasem na niższym poziomie okazuje się, że tworzenie wątków to operacja kosztowna i warto by utworzonymi wątkami zarządzać. *Z punktu widzenia implementacji* rozróżnianie wątków i zadań jest więc zasadne. Do tego wielowątkowość w Javie jest oparta na niskopoziomowym podejściu zapożyczonym jeszcze z języka C, gdzie programista jest uświadamiany i angażowany do detali. Część tej niskopoziomowości przeniknęła do Javy, więc aby pozostać na wysokim poziomie abstrakcji (pożądanym z punktu widzenia projektowego), trzeba przy pisaniu kodu zachować pewną dyscyplinę (którą omówię w dalszej części rozdziału).

W dalszym omówieniu wszędzie tam, gdzie będę opisywał pracę do wykonania, będę stosował pojęcie „zadania"; o „wątkach" będzie zaś mowa jedynie tam, gdzie będzie mi chodzić o konkretny mechanizm wykonawczy zadania. Omawiając model na poziomie koncepcyjnym, należałoby więc mówić o „zadaniach", pomijając ów mechanizm wykonawczy milczeniem.

Łączenie wątków

Wątek może wywołać metodę `join()` innego wątku, co sprawi, że realizacja wątku wywołującego zostanie wznowiona dopiero po zakończeniu wątku, którego metoda `join()` została wywołana. Jeśli jeden wątek wywoła metodę `t.join()` innego wątku `t`, to realizacja wątku wywołującego zostanie wstrzymana aż do czasu zakończenia wątku `t` (czyli do momentu gdy wywołanie `t.isAlive()` zwróci `false`).

Metodę `join()` można także wywołać, podając argument określający czas oczekiwania (wyrażony w milisekundach lub milisekundach i nanosekundach). Dzięki temu, nawet jeśli wątek docelowy nie zostanie skończony w podanym czasie, to wywołanie metody `join()` i tak się zakończy.

Wywołanie metody `join()` można przerwać, wywołując metodę `interrupt()` wątku wywołującego, a zatem konieczne jest zastosowanie klauzuli `try` i `catch`.

Wszystkie opisane wcześniej opcracje zostały przedstawione w poniższym przykładzic:

```
//: concurrency/Joining.java
// O co chodzi w join()...
import static net.mindview.util.Print.*;

class Sleeper extends Thread {
  private int duration;
  public Sleeper(String name, int sleepTime) {
    super(name);
    duration = sleepTime;
    start();
  }
  public void run() {
    try {
      sleep(duration);
    } catch(InterruptedException e) {
      print(getName() + " został przerwany. " +
        "isInterrupted(): " + isInterrupted());
      return;
    }
    print(getName() + " został wybudzony");
  }
}

class Joiner extends Thread {
  private Sleeper sleeper;
  public Joiner(String name, Sleeper sleeper) {
    super(name);
    this.sleeper = sleeper;
    start();
  }
  public void run() {
   try {
      sleeper.join();
    } catch(InterruptedException e) {
      print("Przerwano");
    }
```

```
    print(getName() + " zakończył połączenie");
  }
}

public class Joining {
  public static void main(String[] args) {
    Sleeper
      sleepy = new Sleeper("Śpioch", 1500),
      grumpy = new Sleeper("Gapcio", 1500);
    Joiner
      dopey = new Joiner("Gburek", sleepy),
      doc = new Joiner("Wesołek", grumpy);
    grumpy.interrupt();
  }
} /* Output:
Gapcio został przerwany. isInterrupted(): false
Wesołek zakończył połączenie
Śpioch został wybudzony
Gburek zakończył połączenie
*///:~
```

`Sleeper` to wątek usypiany na czas podany w wywołaniu konstruktora. W metodzie `run()` wywołanie metody `sleep()` może zostać zakończone po upłynięciu podanego czasu, jednak może także zostać przerwane. Wewnątrz klauzuli `catch` wyświetlane są informacje o fakcie przerwania metody `sleep()` oraz o wartości zwracanej przez metodę `isInterrupted()`. Gdy jakiś inny wątek wywoła metodę `interrupt()` danego wątku, to jest w nim ustawiana specjalna flaga. Niemniej jednak flaga ta jest czyszczona w momencie przechwycenia wyjątku, a zatem wewnątrz klauzuli `catch` jej wartość zawsze będzie wynosić `false`. Jest ona wykorzystywana w sytuacjach, gdy wątek może sam sprawdzać, czy wykonywana operacja została przerwana, przy czym sprawdzenie to nie odbywa się w ramach obsługi przerwania.

`Joiner` jest zadaniem, które poprzez wywołanie metody `join()` wątku `Sleeper` oczekuje na jego zakończenie. W metodzie `main()` każdemu wątkowi `Sleeper` odpowiada zadanie `Joiner`. Wyniki generowane przez program pozwalają stwierdzić, że jeśli wątek `Sleeper` zostanie przerwany lub zakończy się w normalny sposób, to wraz z nim zakończy się wykonywanie zadania `Joiner`.

Pamiętaj, że w Javie SE5 biblioteki *java.util.concurrent* zawierają takie narzędzia jak klasa `CyclicBarrier` (prezentowana w dalszej części rozdziału), która może okazać się lepsza od wywołania `join()` pochodzącego z pierwotnej implementacji biblioteki wielowątkowości.

Tworzenie reaktywnego interfejsu użytkownika

Jak już powiedziałem wcześniej, jednym z powodów stosowania wątków jest tworzenie wrażliwego interfejsu użytkownika. Choć *graficznym* interfejsem użytkownika zajmiemy się dopiero w rozdziale „Graficzne interfejsy użytkownika", tutaj zamieściłem przykład interfejsu bazującego na konsoli. Przykład został utworzony w dwóch wersjach. Pierwsza z nich „zawiesza się", realizując długie obliczenia, i nigdy nie odczytuje informacji wpisywanych na konsoli, a druga realizuje te same obliczenia w osobnym zadaniu, dzięki czemu podczas ich wykonywania może także odbierać informacje wpisywane na konsoli.

```
//: concurrency/ResponsiveUI.java
// Reaktywność interfejsu użytkownika.
// {RunByHand}

class UnresponsiveUI {
  private volatile double d = 1;
  public UnresponsiveUI() throws Exception {
    while(d > 0)
      d = d + (Math.PI + Math.E) / d;
    System.in.read(); // Sterowanie nigdy tu nie dotrze
  }
}

public class ResponsiveUI extends Thread {
  private static volatile double d = 1;
  public ResponsiveUI() {
    setDaemon(true);
    start();
  }
  public void run() {
    while(true) {
      d = d + (Math.PI + Math.E) / d;
    }
  }
  public static void main(String[] args) throws Exception {
    //! new UnresponsiveUI(); // Trzeba "ubić" ten proces
    new ResponsiveUI();
    System.in.read();
    System.out.println(d); // Pokazuje postęp operacji
  }
} ///:~
```

Klasa `UnresponsiveUI` wykonuje obliczenia wewnątrz nieskończonej pętli `while`, dlatego niewątpliwie nigdy nie wykona instrukcji odczytującej dane z konsoli (kompilator uważa, że instrukcja ta może zostać wykonana, gdyż jest wyprowadzany w pole poprzez użycie warunku do sterowania pętlą). Jeśli wiersz tworzący obiekt `UnresponsiveUI` zostanie odkomentowany, to aby zakończyć program, konieczne będzie przerwanie („zabicie") procesu.

Aby zapewnić możliwość reakcji programu, należy umieścić obliczenia w metodzie `run()`, którą można wywłaszczyć. Dzięki temu, naciskając klawisz *Enter*, można się przekonać, że obliczenia faktycznie są realizowane w tle, podczas gdy program czeka na informacje wprowadzane przez użytkownika.

Grupy wątków

Grupa wątków przechowuje kolekcję wątków. Wartość grup wątków najlepiej można podsumować, przedstawiając cytat z książki Joshuy Blocha[8], architekta oprogramowania, który pracując dla firmy Sun, poprawił i w znaczącym stopniu usprawnił bibliotekę kontenerów w Javie 1.2 (między innymi):

> *„Grupy wątków najlepiej potraktować jako nieudany eksperyment, a ich istnienie można po prostu zignorować."*

[8] *Effecitive Java™ Programming Language Guide*, Joshua Bloch, Addison-Wesley 2001, strona 211.

Jeśli (jak ja) poświęciłeś trochę czasu i wysiłku na próby poznania wartości grup wątków, zapytasz, dlaczego firma Sun nieco wcześniej nic wydała na ten temat żadnego oficjalnego oświadczenia (to samo pytanie można zadać w przypadku dowolnych zmian, jakie w ciągu lat wprowadzono w języku Java). Joseph Stiglitz, laureat nagrody Nobla w dziedzinie ekonomii, wyznaje filozofię życiową, którą można by zastosować w tym przypadku[9]. Nosi ona nazwę *teorii wzrastającego zaangażowania*:

> *„Koszty naszego trwania w błędzie ponoszą inni, koszty przyznawania się do nich ponosimy my sami."*

Przechwytywanie wyjątków

Z racji natury wątków nie można przechwycić wyjątku, który wyszedł poza wątek. Kiedy wyjątek wydostanie się poza metodę `run()` zadania, jest propagowany do konsoli, chyba że podejmiemy specjalne działania, aby go przyskrzynić. W wersjach poprzedzających SE5 do przechwytywania takich wyjątków wykorzystywało się grupy wątków, ale w Javie SE5 problem rozwiązują wykonawcy, dzięki którym *jakakolwiek* wiedza o grupach wątków staje się zbędna (przydaje się już tylko przy analizie kodu przeznaczonego dla wcześniejszych wersji języka — o grupach wątków możesz poczytać w drugim wydaniu *Thinking in Java*).

Oto zadanie, które zawsze zgłasza wyjątek wymykający się poza metodę `run()`, i metoda `main()`, która ilustruje efekt uruchomienia takiego zadania:

```
//: concurrency/ExceptionThread.java
// {ThrowsException}
import java.util.concurrent.*;

public class ExceptionThread implements Runnable {
  public void run() {
    throw new RuntimeException();
  }
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(new ExceptionThread());
  }
} ///:~
```

Wyjście programu (po wycięciu nadmiarowych kwalifikatorów) wygląda tak:

```
Exception in thread "pool-1-thread-1" java.lang.RuntimeException
        at ExceptionThread.run(ExceptionThread.java:7)
        at ThreadPoolExecutor$Worker.runTask(Unknown Source)
        at ThreadPoolExecutor$Worker.run(Unknown Source)
        at java.lang.Thread.run(Unknown Source)
```

Ujęcie ciała metody `main()` w bloku chronionym niczego nie zmieni:

```
//: concurrency/NaiveExceptionHandling.java
// {ThrowsException}
import java.util.concurrent.*;
```

[9] Oraz w wielu innych przypadkach związanych z doświadczeniami z Javą. Cóż, ale dlaczego się ograniczać? — konsultowałem wiele innych projektów, do których tę zasadę także można zastosować.

```
public class NaiveExceptionHandling {
  public static void main(String[] args) {
    try {
      ExecutorService exec =
        Executors.newCachedThreadPool();
      exec.execute(new ExceptionThread());
    } catch(RuntimeException ue) {
      // Ta instrukcja się NIE wykona!
      System.out.println("Obsłużono wyjątek!");
    }
  }
} ///:~
```

Efekt będzie identyczny, jak w poprzednim przykładzie, a wyjątek pozostanie nieobsłużony.

Aby rozwiązać problem, należałoby zmienić sposób, w jaki klasa Executor tworzy wątki. Otóż w Javie SE5 dostępny jest nowy interfejs Thread.UncaughtExceptionHandler, pozwalający na skojarzenie z obiektem klasy Thread własnej procedury obsługi wyjątku. Kiedy wątek ma zostać zlikwidowany z powodu nieobsłużenia wyjątku, wywoływana jest automatycznie metoda Thread.UncaughtExceptionHandler.uncaughtException(). Aby z niej skorzystać, powinniśmy utworzyć nowy podtyp ThreadFactory, który z każdym obiektem Thread kojarzyłby nowy egzemplarz Thread.UncaughtExceptionHandler. Nowo powstałą wytwórnię wątków przekazywalibyśmy do metody obiektu-wykonawcy tworzącej nowy obiekt usługi wykonawczej ExecutorService:

```
//: concurrency/CaptureUncaughtException.java
import java.util.concurrent.*;

class ExceptionThread2 implements Runnable {
  public void run() {
    Thread t = Thread.currentThread();
    System.out.println("wywołanie run() dla " + t);
    System.out.println(
      "pow = " + t.getUncaughtExceptionHandler());
    throw new RuntimeException();
  }
}

class MyUncaughtExceptionHandler implements
Thread.UncaughtExceptionHandler {
  public void uncaughtException(Thread t, Throwable e) {
    System.out.println("przechwycono " + e);
  }
}

class HandlerThreadFactory implements ThreadFactory {
  public Thread newThread(Runnable r) {
    System.out.println(this + " tworzy nowy obiekt Thread");
    Thread t = new Thread(r);
    System.out.println("utworzono " + t);
    t.setUncaughtExceptionHandler(
      new MyUncaughtExceptionHandler());
    System.out.println(
      "pow = " + t.getUncaughtExceptionHandler());
```

```
    return t;
  }
}

public class CaptureUncaughtException {
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool(
      new HandlerThreadFactory());
    exec.execute(new ExceptionThread2());
  }
} /* Output: (90% match)
HandlerThreadFactory@de6ced tworzy nowy obiekt Thread
utworzono Thread[Thread-0,5,main]
pow = MyUncaughtExceptionHandler@1fb8ee3
run() by Thread[Thread-0,5,main]
pow = MyUncaughtExceptionHandler@1fb8ee3
przechwycono java.lang.RuntimeException
*///:~
```

Posłużyliśmy się dodatkowym śledzeniem mającym na celu sprawdzenie, czy wątki tworzone w wytwórni faktycznie otrzymują nowe egzemplarze `UncaughtExceptionHandler`. Jak widać, nieprzechwycone wyjątki wątku są tym razem przechwytywane w metodzie `uncaughtException()`.

Powyższy przykład pozwala na ustawienie procedury obsługi wyjątków dla poszczególnych wątków z osobna. Jeśli natomiast do wszelkich wyjątków można wydzielić wspólną procedurę obsługi, prościej ustanowić instytucję *domyślnej* procedury obsługi nieprzechwyconych wyjątków wątków, co sprowadza się do ustawienia pola statycznego w klasie `Thread`:

```
//: concurrency/SettingDefaultHandler.java
import java.util.concurrent.*;

public class SettingDefaultHandler {
  public static void main(String[] args) {
    Thread.setDefaultUncaughtExceptionHandler(
      new MyUncaughtExceptionHandler());
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(new ExceptionThread());
  }
} /* Output:
przechwycono java.lang.RuntimeException
*///:~
```

Domyślna procedura obsługi wyjątku jest stosowana jedynie wtedy, kiedy z wątkiem nie skojarzymy żadnej dedykowanej mu procedury. System sprawdza najpierw obecność takiej procedury, a jeśli jej nie znajdzie, kontroluje, czy grupa wątków specjalizuje metodę `uncaughtException()`; jeśli nie, decyduje się na wywołanie `defaultUncaughtExceptionHandler()`.

Współdzielenie zasobów

Program jednowątkowy można traktować jako samotną jednostkę przemieszczającą się w przestrzeni zadań i wykonującą jedną rzecz naraz. Ponieważ istnieje tylko jedna jednostka, nigdy nie trzeba się martwić sytuacją, że dwie w tym samym czasie spróbują wykorzystać ten sam zasób, jak dwoje ludzi próbujących zaparkować w tym samym miejscu lub przejść przez drzwi w tej samej chwili albo nawet jednocześnie mówić.

W przypadku współbieżności samotność już nie istnieje, zachodzi za to możliwość, że dwa lub więcej wątków będzie usiłowało użyć tego samego zasobu równocześnie. Musimy zapobiec kolizji przy dostępie do zasobu albo będziemy mieli dwa wątki próbujące sięgać do tego samego konta bankowego równocześnie, drukować na tej samej drukarce albo modyfikować tę samą zmienną itp.

Niewłaściwy dostęp do zasobów

Przeanalizujmy poniższy przykład, w którym jedno zadanie generuje liczby parzyste, a inne zadanie przetwarza te liczby. Jedynym zadaniem konsumenta liczb jest sprawdzanie ich parzystości.

Na początek zdefiniujemy zadanie konsumenta `EvenChecker`, bo przyda się nam ono w kilku kolejnych przykładach. Aby oddzielić zadanie `EvenChecker` od rozmaitych generatorów, z którymi będziemy go wypróbowywać, utworzymy abstrakcyjną klasę generatora o nazwie `IntGenerator` zawierającą minimum metod potrzebnych zadaniu `EvenChecker`; chodzi mianowicie o metodę `next()` (zwracającą następną wartość parzystą) i możliwość odwołania generacji. Klasa ta nie implementuje interfejsu `Generator`, bo musi wygenerować wartość typu `int`, a uogólnienia nie przyjmują parametrów typowych w postaci typów podstawowych.

```
//: concurrency/IntGenerator.java

public abstract class IntGenerator {
  private volatile boolean canceled = false;
  public abstract int next();
  // Możliwość odwołania generatora:
  public void cancel() { canceled = true; }
  public boolean isCanceled() { return canceled; }
} ///:~
```

Klasa `IntGenerator` posiada metodę `cancel()`, która zmienia stan znacznika `canceled` obiektu generatora, oraz metodę `isCanceled()` zwracającą stan tego znacznika. Ponieważ znacznik jest wartością logiczną typu `boolean`, jego stan jest atomowy, co oznacza, że proste operacje na znaczniku, jak przypisania bądź zwracanie wartości, nie mogą być w żaden sposób przerwane; nie ma więc możliwości, aby pole `canceled` zostało tylko połowicznie zmodyfikowane albo połowicznie odczytane. Znacznik `canceled` jest ponadto zadeklarowany jako ulotny (`volatile`), co ma wymusić jego *widoczność*. O atomowości i widoczności powiemy sobie nieco później.

Klasa EvenChecker może testować dowolny podtyp IntGenerator:

```
//: concurrency/EvenChecker.java
import java.util.concurrent.*;

public class EvenChecker implements Runnable {
  private IntGenerator generator;
  private final int id;
  public EvenChecker(IntGenerator g, int ident) {
    generator = g;
    id = ident;
  }
  public void run() {
    while(!generator.isCanceled()) {
      int val = generator.next();
      if(val % 2 != 0) {
        System.out.println(val + " nieparzysta!");
        generator.cancel(); // Odwołuje wszystkie
                            // egzemplarze EvenChecker
      }
    }
  }
  // Testowanie dowolnego podtypu IntGenerator:
  public static void test(IntGenerator gp, int count) {
    System.out.println("Aby zakończyć, naciśnij Ctrl+C");
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < count; i++)
      exec.execute(new EvenChecker(gp, i));
    exec.shutdown();
  }
  // Domyślna liczba iteracji:
  public static void test(IntGenerator gp) {
    test(gp, 10);
  }
} ///:~
```

Zauważ, że w tym przykładzie klasa dająca się odwołać nie implementuje interfejsu Runnable. Wszystkie zadania EvenChecker, które są zależne od generatora IntGenerator, sprawdzają, czy nie został on odwołany, co widać w metodzie run(). W ten sposób zadania dzielące wspólny zasób (IntGenerator) obserwują ten zasób w oczekiwaniu sygnału zakończenia pracy. Eliminuje to tak zwane sytuacje hazardowe, w których pewna liczba zadań podejmuje jednocześnie reakcję na okoliczności, przez co ze sobą kolidują albo dają niespójne wyniki. Jak widać, każdy aspekt systemu współbieżnego musi być starannie przemyślany i zabezpieczony. Na przykład zadanie nie może zależeć od wykonania innego zadania, bo nie można zagwarantować kolejności kończenia zadań. Tu uzależniamy zadania od obiektu niebędącego zadaniem, eliminując potencjalne sytuacje hazardowe.

Metoda test() przygotowuje i realizuje test dowolnego podtypu IntGenerator, uruchamiając pewną liczbę zadań EvenChecker korzystających z tego samego egzemplarza generatora. Jeśli IntGenerator spowoduje błąd, metoda test() poinformuje o nim i zakończy działanie; w innym przypadku zakończenie programu wymaga skorzystania z kombinacji klawiszy *Ctrl+C*.

Zadanie EvenChecker na okrągło pobiera i sprawdza wartości z generatora. Jeśli `generator.isCanceled()` daje wartość `true`, metoda `run()` zadania kończy działanie, co jest sygnałem dla wykonawcy (w `EvenChecker.test()`), że zadanie dobiegło końca. Każde z zadań EvenChecker może wywołać metodę `cancel()`, odwołując generator, co spowoduje płynne zakończenie pracy przez wszystkie zadania korzystające z tego generatora. W dalszych podrozdziałach przekonasz się, że Java udostępnia bardziej ogólne mechanizmy przerywania wątków.

Pierwsza badana przez nas implementacja `IntGenerator` będzie w metodzie `next()` generować szeregi liczb parzystych:

```
//: concurrency/EvenGenerator.java
// Kiedy wątki wchodzą sobie w drogę.

public class EvenGenerator extends IntGenerator {
  private int currentEvenValue = 0;
  public int next() {
    ++currentEvenValue; // Niebezpieczny punkt!
    ++currentEvenValue;
    return currentEvenValue;
  }
  public static void main(String[] args) {
    EvenChecker.test(new EvenGenerator());
  }
} /* Output: (Sample)
Aby zakończyć, naciśnij Ctrl+C
89476993 nieparzysta!
89476993 nieparzysta!
*///:~
```

Okazuje się, że jedno zadanie może wywołać metodę `next()` generatora tuż po tym, kiedy inne zadanie wymusi pierwszą inkrementację licznika `currentEvenValue`, ale jeszcze przed drugą inkrementacją (w miejscu oznaczonym komentarzem „Niebezpieczny punkt!"). W takim układzie generator zwróci wartość nieparzystą. Aby dowieść takiej możliwości, metoda `EvenChecker.test()` tworzy grupę zadań EvenChecker, które na okrągło pobierają wartości z generatora `EvenGenerator` i sprawdzają ich parzystość. Wykrycie wartości nieparzystej powoduje wypisanie komunikatu i zamknięcie programu.

Prędzej czy później program zawiedzie, bo któreś z zadań EvenChecker odwoła się do generatora, kiedy ten będzie pozostawał w stanie „niepoprawnym". Problem może jednak pozostać niewykryty przez znaczną liczbę cykli kontroli parzystości — ich liczba będzie zależna od cech systemu operacyjnego i innych szczegółów implementacyjnych. Jeśli chcesz przyspieszyć wystąpienie problemu, spróbuj pomiędzy pierwszą a drugą instrukcją inkrementacji licznika generatora umieścić wywołanie metody `yield()`. Ilustruje to zasadniczy kłopot z programami wielowątkowymi — mogą zdawać się zupełnie poprawne i poprawnie działać przez długi czas, mimo obecności błędu.

Wypada zaznaczyć, że sama operacja inkrementacji obejmuje kilka kroków i zadanie może zostać zawieszone przez podsystem wielowątkowości w trakcie inkrementacji — inkrementacja nie jest bowiem w Javie operacją atomową („niepodzielną"). Więc nawet pojedyncze inkrementacje nie stanowią gwarancji zachowania poprawnego stanu zasobu, do którego odwołuje się wiele zadań.

Rozstrzyganie współzawodnictwa o zasoby współdzielone

Poprzedni przykład uwidocznił zasadniczy problem związany ze stosowaniem wątków: nigdy nie wiadomo, kiedy wątek zostanie uruchomiony. Wyobraź sobie, że siedzisz przy stole z widelcem w dłoni i właśnie masz zamiar nabić na niego ostatni kawałek obiadu, lecz gdy sięgasz po niego, obiad znika (gdyż Twój wątek został zawieszony, a w tym czasie jakiś inny ukradł Ci obiad). Tak właśnie wygląda problem, z którym mamy tu do czynienia. Aby współbieżność działała na naszą korzyść, trzeba jakoś zapobiec równoczesnym odwołaniom do wspólnych zasobów, przynajmniej w krytycznych momentach tych odwołań.

Zapobieganie tego rodzaju kolizjom jest po prostu kwestią blokowania (zamknięcia) zasobu, kiedy jakieś zadanie już go używa. Pierwsze zadanie, który sięgnie do zasobu, zamknie go i wtedy inne zadania nie mają doń dostępu, aż nie zostanie odblokowany. Wtedy znowu inne zadanie zablokuje go i wykorzysta itd. Jeśli przednie siedzenie w samochodzie byłoby naszym zasobem, to dziecko, które krzyknie: „Zamawiam!", zablokuje zasób.

Aby rozwiązać problem takich kolizji, niemal we wszystkich schematach współbieżności wykorzystywane jest *szeregowanie dostępu do zasobów wspólnych*. Oznacza to, że w danej chwili tylko jedno zadanie może mieć dostęp do wspólnego zasobu. Zazwyczaj jest to realizowane poprzez umieszczenie fragmentu kodu wewnątrz klauzuli blokującej, dzięki której w danej chwili tylko jedno zadanie może realizować dany kod. Ponieważ klauzula taka powoduje wzajemne wykluczanie zadań z wykonania chronionego nią kodu (ang. *mutual exclusion*), określana jest często jako *mutex*.

Wyobraź sobie łazienkę w domu: wiele osób (zadań wykonywanych w wątkach) może chcieć z niej skorzystać (łazienka jest zatem zasobem współdzielonym). Aby uzyskać dostęp do łazienki, każda z osób puka do drzwi, by sprawdzić, czy jest ona wolna. Jeśli tak, osoba wchodzi do łazienki i zamyka drzwi na klucz. Każde inne zadanie, które chce skorzystać z łazienki, jest „blokowane" i nie może tego zrobić, zatem musi czekać pod drzwiami, aż do momentu gdy łazienka znowu będzie wolna.

Powyższa analogia nie jest już tak trafna, gdy dochodzi do zwolnienia łazienki i przydzielenia jej kolejnemu zadaniu. Zadania nie ustawiają się w kolejce jak ludzie, a wyników działania mechanizmu zarządzającego, jako niedeterministycznych, nie można być pewnym: nie wiadomo, które zadanie uzyska dostęp do łazienki jako następne. Można to sobie wyobrazić inaczej, jak gdyby przed drzwiami do łazienki przepychała się cała grupa zablokowanych chwilowo zadań; kiedy zadanie okupujące łazienkę zwolni blokadę i otworzy drzwi, dostanie się do niej to zadanie, które akurat znajdzie się najbliżej drzwi. Jako się rzekło, można sugerować mechanizmowi zarządzającemu wątkami pewne sposoby działania, wywołując metody `yield()` oraz `setPriority()`, jednak efekty tych sugestii mogą być różne w różnych systemach operacyjnych i implementacjach wirtualnej maszyny Javy.

Java dysponuje wbudowanym mechanizmem zapobiegającym występowaniu kolizji podczas dostępu do zasobów współdzielonych; jest nim słowo kluczowe `synchronized`.

Gdy wątek wyraża chęć wykonania fragmentu kodu chronionego słowem kluczowym synchronized, mechanizm zabezpieczający sprawdza, czy blokada jest wolna, a następnie zakłada blokadę, wykonuje kod i zwalnia blokadę.

Zazwyczaj współdzielony zasób jest fragmentem pamięci reprezentowanym jako obiekt, może się jednak zdarzyć, że będzie to coś innego: plik, port wejścia-wyjścia bądź drukarka. Aby kontrolować dostęp do zasobu współdzielonego, należy go w pierwszej kolejności umieścić wewnątrz obiektu. Następnie należy synchronizować każdą metodę korzystającą z tego zasobu. Oznacza to, że jeśli pewne zadanie wykonuje metodę synchronizowaną, to wszelkie inne zadania próbujące wykonać jakąkolwiek inną metodę synchronizowaną danego obiektu są blokowane do czasu zakończenia wykonania.

Wiesz już, że w kodzie produkcyjnym należałoby oznaczać wszelkie elementy danych klas jako prywatne i ewentualnie udostępniać je za pośrednictwem metod. Synchronizacja tych metod zapobiega zaś kolizjom. Deklaracja taka wygląda następująco:

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

Każdy obiekt zawiera pojedynczą blokadę (zwaną także *monitorem*). Wywołując metodę synchronizowaną, powodujemy, że blokada obiektu jest zajmowana i żadna inna jego synchronizowana metoda nie może zostać wywołana, zanim pierwsza się nie zakończy i nie zwolni blokady. W powyższym przykładzie, jeżeli w jednym zadaniu nastąpi wywołanie metody `f()` na rzecz pewnego obiektu, żadne inne zadanie nie będzie mogło wywołać metody `f()` ani metody `g()` na rzecz tegoż obiektu — aż do czasu, kiedy dojdzie do zakończenia wykonania metody `f()` przez pierwsze zadanie i tym samym do zwolnienia blokady. Tak więc mamy pojedynczą blokadę współdzieloną przez wszystkie synchronizowane metody danego obiektu i to ona zapobiega zapisowi do wspólnej pamięci przez więcej niż jedną metodę równocześnie (tj. nie więcej niż przez jeden wątek równocześnie).

Zauważ, że oznaczanie pól jako prywatnych jest w kontekście współbieżności szczególnie istotne; publiczność pól zniweczy przydatność słowa kluczowego `synchronized` do synchronizacji dostępu do danych obiektu.

Jedno zadanie może pozyskiwać blokadę obiektu wiele razy. Może się to zdarzyć, gdy jedna metoda wywoła inną metodę tego samego obiektu, druga metoda wywoła trzecią itd. Wirtualna maszyna Javy śledzi, ile razy obiekt został zablokowany. Jeśli obiekt jest odblokowany, liczba ta wynosi zero. Gdy zadanie uzyskuje blokadę po raz pierwszy, liczbie blokad przypisywana jest wartość jeden. Jest ona następnie inkrementowana za każdym razem, gdy zadanie uzyskuje kolejną blokadę. Oczywiście takie wielokrotne blokowanie może realizować tylko to zadanie, które założyło pierwszą blokadę. Za każdym razem, gdy zadanie kończy wykonywanie metody synchronizowanej, liczba blokad jest pomniejszana o jeden. Proces ten kończy się w chwili, gdy liczba blokad wyniesie zero — wtedy obiekt przestanie być zablokowany i stanie się dostępny dla innych zadań.

Istnieje także jedna taka blokada dla każdej klasy (część obiektu `Class` odpowiadającego danej klasie), a zatem statyczne metody synchronizowane (deklarowane jako `synchronized static`) mogą blokować się wzajemnie przed równoczesnym dostępem do statycznych danych klasowych.

Kiedy synchronizować metody? Najlepiej posłużyć się regułą synchronizacji według Briana Goetza[10]:

> *Jeśli zapisujesz zmienną, która może być odczytywana przez inny wątek, albo odczytujesz zmienną, która może być zapisywana przez inny wątek, powinieneś zastosować synchronizację; co więcej, zarówno zapisujący, jak i odczytujący powinni synchronizować dostęp za pomocą tej samej blokady.*

Jeśli w klasie występuje więcej metod obsługujących krytyczne dane współdzielone, należy wszystkie je objąć stosowną synchronizacją. Jeśli synchronizowana będzie tylko jedna z tych metod, pozostałe będą ignorować niedostępność blokady i będą mogły być bez przeszkód wywoływane. To bardzo ważne: każda metoda, która ma dostęp do zasobu współdzielonego, musi być synchronizowana (albo nici z synchronizacji dostępu).

Synchronizacja klasy EvenGenerator

Synchronizując metody klasy `EvenGenerator`, możemy uchronić się przed niepożądanymi efektami współbieżnego działania wątków:

```
// : concurrency/SynchronizedEvenGenerator.java
// Uproszczenie muteksów za pomocą słowa kluczowego synchronized.
// {RunByHand}

public class
SynchronizedEvenGenerator extends IntGenerator {
  private int currentEvenValue = 0;
  public synchronized int next() {
    ++currentEvenValue;
    Thread.yield(); // Przyspieszone sprowokowanie błędu
    ++currentEvenValue;
    return currentEvenValue;
  }
  public static void main(String[] args) {
    EvenChecker.test(new SynchronizedEvenGenerator());
  }
} ///:~
```

Pomiędzy instrukcjami inkrementacji licznika pojawiło się wywołanie `Thread.yield()`, które znacznie zwiększa prawdopodobieństwo przełączenia kontekstu w czasie, kiedy wartość licznika jest niewłaściwa. Mimo to, ponieważ jednak muteks zapobiega wykonywaniu krytycznej operacji przez więcej niż jedno zadanie, nie dochodzi do błędu; samo wywołanie `yield()` można potraktować jako sposób na szybsze wykrycie ewentualnych kolizji, które normalnie mogą długo pozostać nieujawnione.

Pierwsze zadanie, które wejdzie do metody `next()`, założy blokadę, przez co następne zadania próbujące tego samego będą blokowane aż do czasu zwolnienia blokady przez pierwsze zadanie. W tym momencie planista wybierze spośród oczekujących następne zadanie. W efekcie kod chroniony muteksem będzie zawsze wykonywany w ramach tylko jednego zadania.

[10] Brian Goetz, współautor *Java Concurrency in Practice* (Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes i Doug Lea, Addison-Wesley, 2006).

Ćwiczenie 11. Utwórz klasę zawierającą dwa pola danych i metodę, która manipuluje wartością tych pól w wieloetapowych operacjach, tak aby w międzyczasie pola te miały „niepoprawną" wartość (wedle dowolnie przyjętego kryterium poprawności). Dodaj metody odczytujące pola i utwórz wiele wątków wywołujących te metody; pokaż, że dane są niekiedy widoczne w stanie „niepoprawnym". Usuń problem, stosując słowo kluczowe `synchronized` (3).

Blokady jawne — obiekty Lock

Biblioteka `java.util.concurrent` w Javie SE5 zawiera także mechanizm muteksów jawnych definiowanych w pakiecie `java.util.concurrent.locks`. Jego rdzeń, w postaci obiektu `Lock`, musi być tworzony jawnie przez zadanie i jawnie blokowany oraz odblokowywany; kod wykorzystujący jawne blokady jest więc mniej elegancki niż kod korzystający z synchronizacji wbudowanej. Ten pierwszy jest za to bardziej elastyczny i lepiej sprawdza się w rozwiązaniach niektórych problemów. Oto przykład *SynchronizedEvenGenerator.java* przepisany z użyciem jawnych blokad `Lock`:

```
//: concurrency/MutexEvenGenerator.java
// Zapobieganie kolizjom wątków przy użyciu jawnych blokad.
// {RunByHand}
import java.util.concurrent.locks.*;

public class MutexEvenGenerator extends IntGenerator {
  private int currentEvenValue = 0;
  private Lock lock = new ReentrantLock();
  public int next() {
    lock.lock();
    try {
      ++currentEvenValue;
      Thread.yield(); // Przyspieszenie katastrofy
      ++currentEvenValue;
      return currentEvenValue;
    } finally {
      lock.unlock();
    }
  }
  public static void main(String[] args) {
    EvenChecker.test(new MutexEvenGenerator());
  }
} ///:~
```

Klasa `MutexEvenGenerator` zawiera muteks o nazwie `lock` i wykorzystuje jego metody `lock()` (założenie blokady) i `unlock()` (zwolnienie blokady) do utworzenia sekcji krytycznej w obrębie metody `next()`. Korzystając z obiektów klasy `Lock`, powinniśmy zawsze wdrażać zaprezentowany idiom: zaraz za wywołaniem `lock()` należy umieścić blok kodu chronionego z klauzulą `finally` zawierającą wywołanie `unlock()` — to jedyny sposób zagwarantowania, że blokada zostanie każdorazowo zwolniona. Zauważ, że instrukcja `return` musi pojawić się wewnątrz bloku `try`, aby zapobiec przedwczesnemu wywołaniu `unlock()`, co wyeksponowałoby chronione dane innym zadaniom.

Choć blok `try-finally` wymaga nieco więcej kodu niż wbudowana synchronizacja metod, reprezentuje przy okazji jedną z zalet jawnych blokad. Otóż jeśli w obrębie metody synchronizowanej coś pójdzie nie tak, dojdzie do zgłoszenia wyjątku i nie będziemy mieli

okazji wykonania operacji porządkowych, gwarantujących spójność stanu systemu. Przy zastosowaniu jawnych obiektów Lock owe operacje możemy wykonać właśnie w obrębie klauzuli `finally`.

Z drugiej strony przy wykorzystywaniu wbudowanej synchronizacji metod zyskujemy wygodę programowania i zmniejszamy ilość kodu, co przekłada się też na mniejszą liczbę błędów; z tego powodu jawne blokady Lock powinniśmy stosować wyłącznie do rozwiązywania konkretnych problemów. Na przykład słowo synchronized nie pozwala na wypróbowywanie założenia blokady albo na podejmowanie prób założenia blokady z limitem czasu oczekiwania — w tym celu trzeba skorzystać z biblioteki concurrent:

```
//: concurrency/AttemptLocking.java
// Blokady z biblioteki współbieżności pozwalają na
// rezygnację z przeciągających się prób zakładania blokad.
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AttemptLocking {
  private ReentrantLock lock = new ReentrantLock();
  public void untimed() {
    boolean captured = lock.tryLock();
    try {
      System.out.println("tryLock(): " + captured);
    } finally {
      if(captured)
        lock.unlock();
    }
  }
  public void timed() {
    boolean captured = false;
    try {
      captured = lock.tryLock(2, TimeUnit.SECONDS);
    } catch(InterruptedException e) {
      throw new RuntimeException(e);
    }
    try {
      System.out.println("tryLock(2, TimeUnit.SECONDS): " +
        captured);
    } finally {
      if(captured)
        lock.unlock();
    }
  }
  public static void main(String[] args) {
    final AttemptLocking al = new AttemptLocking();
    al.untimed(); // True -- blokada dostępna
    al.timed();   // True -- blokada dostępna
    // Utworzenie osobnego zadania w celu założenia blokady:
    new Thread() {
      { setDaemon(true); }
      public void run() {
        al.lock.lock();
        System.out.println("założona");
      }
    }.start();
```

```
      Thread.yield(); // Dajmy szansę drugiemu zadaniu
      al.untimed(); // False -- blokada założona przez zadanie
      al.timed();   // False -- blokada założona przez zadanie
    }
  } /* Output:
  tryLock(): true
  tryLock(2, TimeUnit.SECONDS): true
  założona
  tryLock(): false
  tryLock(2, TimeUnit.SECONDS): false
  *///:~
```

Blokada `ReentrantLock` pozwala na podejmowanie kontrolowanych prób założenia blokady (metoda `untimed()`), tak aby w przypadku, gdy ktoś ją wcześniej założył i przetrzymuje, zadanie mogło zrezygnować z nieskutecznych prób i podjąć w międzyczasie inne czynności. Z kolei metoda `timed()` pozwala na podjęcie próby założenia blokady z ograniczeniem czasu oczekiwania — tu do dwóch sekund (zwróć uwagę na użycie klasy `TimeUnit` z Javy SE5 do określenia jednostek czasu). W metodzie `main()` tworzony jest osobny wątek w postaci klasy anonimowej; zakłada on blokadę, tak aby metody `timed()` i `untimed()` mogły się wykazać.

Jawne obiekty blokad dają programiście większy poziom kontroli nad blokowaniem i zdejmowaniem blokad w porównaniu z blokadą wymuszaną słowem kluczowym `synchronized`. Przydaje się to przy implementowaniu specjalizowanych struktur synchronizacji w rodzaju blokad wiązanych — wykorzystywanych do przeglądania węzłów listy (zadanie przeglądające węzły musi założyć blokadę następnego węzła, zanim zwolni blokadę węzła bieżącego).

Atomowość i widoczność

W dyskusjach na temat wielowątkowości w Javie często przewija się niepoprawne twierdzenie, że: „Operacje atomowe nie muszą być synchronizowane". Otóż *operacja atomowa* (inaczej: „niepodzielna") to czynność, której mechanizm zarządzający wątkami nie jest w stanie przerwać — jeśli operacja się rozpocznie, będzie wykonywana aż do zakończenia, a dopiero potem pojawi się możliwość *zmiany kontekstu*. Jednakże poleganie na niepodzielności operacji jest ryzykowne i niebezpieczne — synchronizację można zastępować atomowością tylko w przypadku posiadania eksperckiej wiedzy w dziedzinie współbieżności albo dysponując pomocą ze strony takiego eksperta. Każdy, kto sądzi, że jest wystarczająco sprytny, żeby chwycić tego byka za rogi, powinien wziąć sobie do serca *regułę Goetza*[11]:

> *Jeśli potrafisz napisać wydajną implementację maszyny wirtualnej dla nowoczesnego mikroprocesora, jesteś wystarczająco wykwalifikowany, aby zacząć myśleć o unikaniu synchronizacji*[12].

[11] Za wspominanym już Brianem Goetzem, ekspertem od współbieżności, który pomógł mi ułożyć materiał do tego rozdziału.

[12] Z tej reguły wynika następna: „Jeśli ktoś uważa, że wielowątkowość jest prosta i oczywista, trzeba się upewnić, że nie ma wpływu na podejmowanie ważnych decyzji w projekcie. Inaczej kłopoty gotowe".

Warto mimo to wiedzieć o atomowości, jak i o tym, że przy użyciu odpowiednio zaawansowanych technik została ona wykorzystana do implementacji niektórych komponentów biblioteki `java.util.concurrent`. Jednakże wystrzegaj się podobnych ciągot: pamiętaj o regule synchronizacji Briana.

Atomowość dotyczy „prostych" operacji na typach podstawowych, z wyjątkiem typów `long` i `double`. Odczyt i zapis zmiennych podstawowych poza `double` i `long` to operacje niepodzielne. Jednak już w przypadku wartości 64-bitowych (właśnie `long` i `double`) maszyna wirtualna Javy może realizować operacje jako pary operacji 32-bitowych, prowokując potencjalną możliwość przełączenia kontekstu w połowie odczytu czy zapisu, co z kolei oznacza ryzyko uwidocznienia innemu zadaniu niepoprawnych wartości (efekt ten określa się czasem jako *„jąkanie"*, bo oznacza widoczność jedynie częściowego efektu operacji). Niemniej jednak można uzyskać efekt „atomowości" (dla operacji przypisania i zwracania wartości), definiując zmienne typów `long` i `double` jako zmienne ulotne, a więc ze słowem `volatile` (słowo to nie działało poprawnie przed wersją SE5). Różne implementacje maszyn wirtualnych mają swobodę dawania nawet szerszych gwarancji niepodzielności, ale wtedy mówimy już o zależności zachowania programu od platformy.

Wiemy już, że operacje atomowe nie mogą być przerwane przez mechanizm wielowątkowości. Programiści będący ekspertami mogą skorzystać z tej cechy niektórych operacji do pisania kodu pozbawionego blokad, którego nie trzeba jawnie synchronizować. Ale byłoby to nadmiernym uproszczeniem. Niekiedy, nawet jeśli się wydaje, że operacja atomowa byłaby bezpieczna, rzeczywistość może okazać się inna. Czytelnicy książki zapewne w większości nie podlegają regule Goetza odnośnie kwalifikacji potrzebnych do podejmowania takich decyzji i nie powinni próbować zastępować synchronizacji atomowością. Próba pozbycia się elementów synchronizacji to przejaw przedwczesnej optymalizacji, który przy niewielkim albo żadnym zysku może mieć katastrofalne konsekwencje.

W systemach wieloprocesorowych (które ostatnio pojawiają się w postaci procesorów wielordzeniowych — zawierających kilka jednostek obliczeniowych we wspólnym układzie) do aspektów atomowości dochodzi kwestia widoczności, znacznie bardziej tu eksponowana niż w systemach jednoprocesorowych. Otóż zmiany wprowadzane przez jedno zadanie, nawet jeśli są atomowe w sensie niepodzielności, mogą być niewidoczne dla innych zadań (zmiany mogą się na przykład odbywać w lokalnej pamięci podręcznej procesora), przez co różne zadania będą różnie postrzegać stan aplikacji. Z drugiej strony mechanizm synchronizacji wymusza, aby zmiany wprowadzane przez zadanie w systemie wieloprocesorowym były widoczne dla wszystkich współbieżnych zadań aplikacji. Bez synchronizacji nie sposób określić momentu uwidocznienia zmiany.

Do zapewniania widoczności wartości w obrębie aplikacji służy słowo kluczowe `volatile`. Jeśli pole obiektu zostanie zadeklarowane jako ulotne (`volatile`), to natychmiast po zapisie wartości w tym polu nowa wartość będzie widoczna dla wszystkich operacji odczytu, niezależnie od wykorzystywania lokalnych pamięci podręcznych — pola `volatile` są natychmiast aktualizowane w pamięci głównej systemu, a odczyty są również wykonywane jedynie z tej pamięci.

Trzeba zdawać sobie sprawę z tego, że atomowość i widoczność to dwie zupełnie różne kwestie. Operacja atomowa na polu nieulotnym niekoniecznie musi wiązać się z natychmiastową aktualizacją pamięci z podręcznego bufora procesora, więc mimo niepodzielności operacji inne zadanie odczytujące wartość pola w systemie wieloprocesorowym może otrzymać poprzednią wartość. Dlatego, jeśli do danego pola odwołuje się więcej niż jedno zadanie, należałoby to pole oznaczyć jako `volatile`; w przeciwnym razie wszelkie odwołania do tego pola trzeba zabezpieczyć przez synchronizację. Synchronizacja również wymusza aktualizację pamięci głównej, więc jeśli pole jest dostępne wyłącznie za pomocą metod synchronizowanych albo chronione jawnymi blokadami, nie trzeba nadawać mu atrybutu ulotności.

Wszelkie zapisy, które wykona dane zadanie, będą natychmiast widoczne dla tego zadania, więc jeśli pole jest wykorzystywane tylko przez jedno zadanie, nie musi być definiowane jako `volatile`.

Słowo `volatile` nie spełnia swojej roli, jeśli wartość pola zależy od jego poprzedniej wartości (na przykład w sytuacji inkrementacji licznika) ani kiedy wartości pola są ograniczane przez wartości innych pól (np. w sytuacji, gdy pola `dolnaGranica` i `gornaGranica` klasy `Zakres` muszą spełniać relację `dolnaGranica <= gornaGranica`).

Słowo `volatile` zamiast słowa kluczowego `synchronized` może być stosowane jedynie w klasach, które posiadają tylko jedno zmienne pole; a i tak w pierwszym podejściu należałoby zastosować jednak słowo `synchronized` — to znacznie bezpieczniejsze podejście, wszystkie inne niosą ze sobą jakieś ryzyko.

Jak rozpoznać operację atomową? Przypisanie wartości do pola i odczytanie tej wartości to zwykle operacje atomowe, ale na przykład w języku C++ atomowe mogą być również poniższe operacje:

```
i++;          // operacja potencjalnie atomowa w C++
i += 2;       // operacja potencjalnie atomowa w C++
```

Jednakże w języku C++ atomowość powyższych operacji zależy od implementacji kompilatora i cech procesora. Nie sposób w C++ napisać kodu, który opierałby się na atomowości operacji i był kodem przenośnym pomiędzy platformami. C++ nie oferuje programistom żadnego spójnego *modelu pamięci*, z którym mamy do czynienia w Javie (w Javie SE5)[13].

W Javie powyższe operacje z całą pewnością nie są atomowe, co można sprawdzić, przyglądając się odpowiadającym im instrukcjom kodu bajtowego wygenerowanego przez kompilator:

```
//: concurrency/Atomicity.java
// {Exec: javap -c Atomicity}

public class Atomicity {
  int i;
  void f1() { i++; }
  void f2() { i += 3; }
```

[13] Ma to się zmienić w przyszłym wydaniu standardu języka C++.

```
} /* Output: (Sample)
...
void f1();
 Code:
  0:    aload_0
  1:    dup
  2:    getfield      #2; //Field i:I
  5:    iconst_1
  6:    iadd
  7:    putfield      #2; //Field i:I
  10:   return

void f2();
 Code:
  0:    aload_0
  1:    dup
  2:    getfield      #2; //Field i:I
  5:    iconst_3
  6:    iadd
  7:    putfield      #2; //Field i:I
  10:   return
*///:~
```

Każda analizowana instrukcja Javy generuje instrukcje „get" i „put" rozdzielone innymi instrukcjami. Tak więc pomiędzy odczytem („get") i zapisem („put") wartości pola może dojść do podjęcia innego zadania, które również operowałoby na tym polu — nie ma więc mowy o atomowości.

Gdybyśmy ślepo zastosowali cechę atomowości, uznalibyśmy, że do definicji tej cechy pasowałaby metoda `getValue()` z poniższego programu:

```
//: concurrency/AtomicityTest.java
import java.util.concurrent.*;

public class AtomicityTest implements Runnable {
  private int i = 0;
  public int getValue() { return i; }
  private synchronized void evenIncrement() { i++; i++; }
  public void run() {
    while(true)
      evenIncrement();
  }
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    AtomicityTest at = new AtomicityTest();
    exec.execute(at);
```

```
      while(true) {
        int val = at.getValue();
        if(val % 2 != 0) {
          System.out.println(val);
          System.exit(0);
        }
      }
    }
  } /* Output: (Sample)
  191583767
  *///:~
```

Ale jak widać, program mimo wszystko doczeka się wartości nieparzystej i przerwie działanie. Choć instrukcja `return i` faktycznie jest operacją atomową, to brak synchronizacji odwołań do wartości sprawia, że wartość może być odczytywana w czasie, kiedy obiekt pozostaje w niepoprawnym stanie pośrednim. Do tego i nie jest deklarowane jako wartość ulotna, co powoduje dodatkowo problemy z widocznością zmian. Z tego powodu tak metoda `getValue()`, *jak i* `evenIncrement()`, powinny być synchronizowane (deklarowane ze słowem `synchronized`). Jeszcze raz: tylko eksperci w dziedzinie współbieżności są wystarczająco wykwalifikowani, żeby w podobnych sytuacjach podejmować próby optymalizacji przez rezygnację z synchronizacji; ponownie postuluję stosowanie reguły synchronizacji Briana.

Jako kolejny przykład przeanalizujmy coś jeszcze prostszego: klasę generującą numer seryjny[14]. Każde wywołanie metody `nextSerialNumber()` musi zwrócić unikalną wartość:

```
//: concurrency/SerialNumberGenerator.java

public class SerialNumberGenerator {
  private static volatile int serialNumber = 0;
  public static int nextSerialNumber() {
    return serialNumber++; // Brak zabezpieczenia na
                           // okoliczność wielowątkowości
  }
} ///:~
```

Klasa `SerialNumberGenerator` jest w zasadzie tak prosta, że trudno sobie wyobrazić coś prostszego; jeśli dodatkowo mamy podstawy wyniesione z języka C++ lub innego języka operującego na niższym poziomie, to moglibyśmy sądzić, że inkrementacja jest operacją atomową, gdyż może być ona implementowana jako pojedyncza instrukcja mikroprocesora. Jednak w wirtualnej maszynie Javy inkrementacja *nie* jest operacją atomową, a jej realizacja wymaga zarówno odczytu, jak i zapisu. Dlatego też, w przypadku wykorzystania wątków, problemy mogą się pojawić nawet w tak prostej klasie. Nie mamy tu kłopotu z widocznością; prawdziwy problem polega na tym, że metoda `nextSerialNumber()` odwołuje się do współdzielonej, zmiennej wartości, zaniedbując synchronizację dostępu.

Pole `serialNumber` zostało oznaczone jako `volatile`, gdyż każdy wątek może mieć lokalny stos i przechowywać na nim kopie niektórych zmiennych. Deklarując zmienną jako `volatile`, informujemy kompilator, by nie optymalizował żadnych operacji odczytu

[14] Przykład został zainspirowany książką Joshuy Blocha pt.: *Effective Java™ Programming Language Guide*, Addison-Wesley, 2001, strona 190.

i zapisu danego pola, przez co możemy zapewnić jego pełną synchronizację z lokalnymi danymi wątku. W efekcie zapisy i odczyty odbywają się w pamięci operacyjnej, a nie pamięci podręcznej procesora. Słowo `volatile` zabrania też kompilatorowi zmian kolejności odwołań w ramach ewentualnych optymalizacji, ale obecność `volatile` nie zmienia faktu, że inkrementacja nie jest w Javie operacją atomową.

Jeśli pole klasy jest dostępne z poziomu wielu zadań, a przynajmniej jeden z tych dostępów polega na zapisie, należy oznaczać takie pole jako ulotne słowem `volatile`. Na przykład pole wykorzystywane jako znacznik przerwania zadania musi być polem `volatile`; inaczej mogłoby dojść do zbuforowania wartości znacznika w rejestrze procesora, a wtedy zmiany tej wartości wprowadzone poza bieżącym zadaniem byłyby niewidoczne dla tego zadania i znacznik nie spełniałby swojej roli.

Do przetestowania powyższej klasy potrzebny nam będzie zbiór, który nie doprowadzi do wyczerpania pamięci w przypadku, gdy sprowokowanie problemów zabierze dużo czasu. Przedstawiony poniżej `CircularSet` wielokrotnie wykorzystuje pamięć używaną do przechowywania liczb całkowitych, gdyż zakładamy, że do czasu rozpoczęcia ponownego zapisywania zbioru prawdopodobieństwo kolizji z wartością nadpisywaną jest minimalne. W celu uniknięcia kolizji podczas pracy wielowątkowej metoda `add()` oraz `contains()` są synchronizowane:

```
// : concurrency/SerialNumberChecker.java
// Operacje pozornie bezpieczne nie są takimi,
// kiedy mamy do czynienia z wątkami.
// {Args: 4}
import java.util.concurrent.*;

// Klasa cyklicznie wykorzystująca przydzieloną pamięć:
class CircularSet {
  private int[] array;
  private int len;
  private int index = 0;
  public CircularSet(int size) {
    array = new int[size];
    len = size;
    // Inicjalizacja wartością nie generowaną
    // przez generator SerialNumberGenerator:
    for(int i = 0; i < size; i++)
      array[i] = -1;
  }
  public synchronized void add(int i) {
    array[index] = i;
    // "Zawinięcie" indeksu i zamazywanie "starych" elementów:
    index = ++index % len;
  }
  public synchronized boolean contains(int val) {
    for(int i = 0; i < len; i++)
      if(array[i] == val) return true;
    return false;
  }
}

public class SerialNumberChecker {
  private static final int SIZE = 10;
  private static CircularSet serials =
```

```
    new CircularSet(1000);
  private static ExecutorService exec =
    Executors.newCachedThreadPool();
  static class SerialChecker implements Runnable {
    public void run() {
      while(true) {
        int serial =
          SerialNumberGenerator.nextSerialNumber();
        if(serials.contains(serial)) {
          System.out.println("Duplikat: " + serial);
          System.exit(0);
        }
        serials.add(serial);
      }
    }
  }
  public static void main(String[] args) throws Exception {
    for(int i = 0; i < SIZE; i++)
      exec.execute(new SerialChecker());
    // Ewentualny argument wstrzymuje wykonanie wątku main():
    if(args.length > 0) {
      TimeUnit.SECONDS.sleep(new Integer(args[0]));
      System.out.println("Nie wykryto duplikatów");
      System.exit(0);
    }
  }
} /* Output: (Sample)
Duplikat: 8468656
*///:~
```

`SerialNumberChecker` zawiera statyczny obiekt `CircularSet` przechowujący wszystkie pobrane numery seryjne oraz zagnieżdżoną klasę `SerialChecker` sprawdzającą unikatowość numerów seryjnych. Tworząc wiele zadań, które współzawodniczą o numery seryjne, w miarę szybko można odkryć, że numery te w końcu powtórzą się, wystarczy odpowiednio długo poczekać. Aby rozwiązać powstały problem, wystarczy poprzedzić metodę `nextSerialNumber()` słowem kluczowym `synchronized`.

Operacjami atomowymi uważanymi za bezpieczne jest odczyt i zapis danych typów podstawowych. Niemniej jednak nie należy na tym zbytnio polegać, gdyż, jak pokazał przykład programu *AtomicityTest.java*, mimo niepodzielności operacji może dojść do sytuacji, w której zostanie ona wykorzystana do odwołania do obiektu pozostającego w nieustalonym stanie przejściowym. Znów wychodzi na to, że najlepiej po prostu stosować regułę synchronizacji Briana.

Ćwiczenie 12. Popraw program *AtomicityTest.java* za pomocą słowa kluczowego `synchronized`. Czy potrafisz udowodnić skuteczność poprawki (3)?

Ćwiczenie 13. Popraw program *SerialNumberChecker.java* za pomocą słowa kluczowego `synchronized`. Czy potrafisz udowodnić skuteczność poprawki (1)?

Klasy „atomowe"

Java SE5 zawiera zestaw specjalnych klas zmiennych atomowych, jak `AtomicInteger`, `AtomicLong`, `AtomicReference` itd., które udostępniają niepodzielną operację warunkowej aktualizacji wartości w postaci metody:

```
boolean compareAndSet(wartośćOczekiwana, wartośćNowa);
```

Klasy te robią użytek z niskopoziomowej atomowości niektórych operacji charakterystycznej dla części współczesnych procesorów, więc w codziennej praktyce programistycznej występują rzadko. Okazjonalnie przydają się w zwykłym kodowaniu, ale też tylko tam, gdzie dochodzi do precyzyjnego strojenia aplikacji pod względem wydajności. Moglibyśmy na przykład przepisać program *AtomicityTest.java* z użyciem klasy `AtomicInteger`:

```
//: concurrency/AtomicIntegerTest.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

public class AtomicIntegerTest implements Runnable {
  private AtomicInteger i = new AtomicInteger(0);
  public int getValue() { return i.get(); }
  private void evenIncrement() { i.addAndGet(2); }
  public void run() {
    while(true)
      evenIncrement();
  }
  public static void main(String[] args) {
    new Timer().schedule(new TimerTask() {
      public void run() {
        System.err.println("Kończę...");
        System.exit(0);
      }
    }, 5000); // Przerwanie po 5 sekundach
    ExecutorService exec = Executors.newCachedThreadPool();
    AtomicIntegerTest ait = new AtomicIntegerTest();
    exec.execute(ait);
    while(true) {
      int val = ait.getValue();
      if(val % 2 != 0) {
        System.out.println(val);
        System.exit(0);
      }
    }
  }
} ///:~
```

Wyeliminowaliśmy słowo kluczowe `synchronized`, stosując w zamian klasę `AtomicInteger`. Ponieważ program w takiej postaci nie doczeka się nigdy błędu, za pomocą klasy `Timer` ustawiany jest limit czasu wykonania programu.

Podobnie można by przepisać program *MutexEvenGenerator.java*:

```
//: concurrency/AtomicEvenGenerator.java
// Klasy atomowe z rzadka przydają się w zwyczajnym kodzie.
// {RunByHand}
```

```
import java.util.concurrent.atomic.*;

public class AtomicEvenGenerator extends IntGenerator {
  private AtomicInteger currentEvenValue =
    new AtomicInteger(0);
  public int next() {
    return currentEvenValue.addAndGet(2);
  }
  public static void main(String[] args) {
    EvenChecker.test(new AtomicEvenGenerator());
  }
} ///:~
```

I tutaj udało się wyeliminować wszelkie formy synchronizacji przez zastosowanie klasy `AtomicInteger`.

Trzeba pamiętać, że klasy atomowe zostały zaprojektowane pod kątem potrzeb implementacji klas z biblioteki `java.util.concurrent` i że we własnym kodzie należy ich używać wstrzemięźliwie, tylko tam, gdzie istnieje pewność co do braku innych problemów. W przypadku ogólnym lepiej jednak polegać na blokadach (tak jawnych, w postaci obiektów `Lock`, jak też narzucanych słowem `synchronized`).

Ćwiczenie 14. Pokaż, że klasa `java.util.Timer` daje się skalować do wielkich liczb, tworząc program generujący wiele obiektów `Timer`, które po upływie czasu oczekiwania wykonują jakieś proste zadania (4).

Sekcje krytyczne

Czasami chcemy jedynie uniemożliwić wielu wątkom jednoczesny dostęp do fragmentu kodu, a nie do całej metody. Sekcje kodu izolowane w taki sposób są nazywane *sekcjami krytycznymi*, a tworzy się je przy użyciu słowa kluczowego `synchronized`. W poniższym przykładzie `synchronized` jest stosowane do wskazania obiektu, którego blokada synchronizuje blok kodu:

```
synchronized(obiektSynchronizujacy) {
    // W danej chwili dostęp do tego fragmentu kodu
    // będzie mieć tylko jeden wątek.
}
```

Taki fragment kodu nazywany jest także *blokiem synchronizowanym*; zanim będzie można go wykonać, konieczne będzie uzyskanie blokady `obiektuSynchronizującego`. Jeśli jakiś inny wątek już posiada blokadę, wejście innego wątku do sekcji krytycznej nie jest możliwe — będzie to mogło nastąpić dopiero po zwolnieniu blokady.

Poniższy przykład porównuje oba sposoby synchronizacji, pokazując, jak zwiększa się czas dostępności obiektu dla innych wątków, w przypadku gdy zamiast całej metody synchronizowany jest jedynie jej fragment. Dodatkowo program pokazuje, w jaki sposób w programach wielowątkowych można używać klasy niechronionej, jeśli jest ona zarządzana i ochraniana przez inną klasę:

```
//: concurrency/CriticalSection.java
// Synchronizowanie bloków kodu zamiast metod z ilustracją
// działania ochrony klasy ignorującej aspekty wielowątkowości
```

```
// przez klasę odpowiednio zabezpieczoną na okoliczność wątków.
package concurrency;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

class Pair { // Niezabezpieczona
  private int x, y;
  public Pair(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public Pair() { this(0, 0); }
  public int getX() { return x; }
  public int getY() { return y; }
  public void incrementX() { x++; }
  public void incrementY() { y++; }
  public String toString() {
    return "x: " + x + ", y: " + y;
  }
  public class PairValuesNotEqualException
  extends RuntimeException {
    public PairValuesNotEqualException() {
      super("Wartości pary są nierówne: " + Pair.this);
    }
  }
  // Niezmiennik -- obie zmienne muszą być równe:
  public void checkState() {
    if(x != y)
      throw new PairValuesNotEqualException();
  }
}

// Ochrona klasy Pair w klasie zabezpieczonej na
// okoliczność pracy wielowątkowej:
abstract class PairManager {
  AtomicInteger checkCounter = new AtomicInteger(0);
  protected Pair p = new Pair();
  private List<Pair> storage =
    Collections.synchronizedList(new ArrayList<Pair>());
  public synchronized Pair getPair() {
    // Utworzenie kopii w celu zabezpieczenia oryginału:
    return new Pair(p.getX(), p.getY());
  }
  // Zakładamy, że to długotrwała operacja
  protected void store(Pair p) {
    storage.add(p);
    try {
      TimeUnit.MILLISECONDS.sleep(50);
    } catch(InterruptedException ignore) {}
  }
  public abstract void increment();
}

// Synchronizacja całej metody:
class PairManager1 extends PairManager {
  public synchronized void increment() {
    p.incrementX();
```

```
    p.incrementY();
    store(getPair());
  }
}

// Zastosowanie sekcji krytycznej:
class PairManager2 extends PairManager {
  public void increment() {
    Pair temp;
    synchronized(this) {
      p.incrementX();
      p.incrementY();
      temp = getPair();
    }
    store(temp);
  }
}

class PairManipulator implements Runnable {
  private PairManager pm;
  public PairManipulator(PairManager pm) {
    this.pm = pm;
  }
  public void run() {
    while(true)
      pm.increment();
  }
  public String toString() {
    return "Para: " + pm.getPair() +
      " licznik = " + pm.checkCounter.get();
  }
}

class PairChecker implements Runnable {
  private PairManager pm;
  public PairChecker(PairManager pm) {
    this.pm = pm;
  }
  public void run() {
    while(true) {
      pm.checkCounter.incrementAndGet();
      pm.getPair().checkState();
    }
  }
}

public class CriticalSection {
  // test obu rozwiązań:
  static void
  testApproaches(PairManager pman1, PairManager pman2) {
    ExecutorService exec = Executors.newCachedThreadPool();
    PairManipulator
      pm1 = new PairManipulator(pman1),
      pm2 = new PairManipulator(pman2);
    PairChecker
      pcheck1 = new PairChecker(pman1),
      pcheck2 = new PairChecker(pman2);
    exec.execute(pm1);
```

```
      exec.execute(pm2);
      exec.execute(pcheck1);
      exec.execute(pcheck2);
      try {
        TimeUnit.MILLISECONDS.sleep(500);
      } catch(InterruptedException e) {
        System.out.println("Uśpienie przerwane");
      }
      System.out.println("pm1: " + pm1 + "\npm2: " + pm2);
      System.exit(0);
    }
    public static void main(String[] args) {
      PairManager
        pman1 = new PairManager1(),
        pman2 = new PairManager2();
      testApproaches(pman1, pman2);
    }
} /* Output: (Sample)
pm1: Para: x: 15, y: 15 licznik = 272565
pm2: Para: x: 16, y: 16 licznik = 3956974
*///:~
```

Zgodnie z tym, co napisałem wcześniej, klasy `Pair` nie można bezpiecznie używać w programach wielowątkowych, gdyż jej niezmiennik wymaga, aby obie zmienne miały identyczne wartości. Co więcej, jak pokazały przykłady przedstawione we wcześniejszej części rozdziału, także operacja inkrementowania nie jest bezpieczna w rozwiązaniach wielowątkowych, a ponieważ żadna z metod klasy `Pair` nie jest synchronizowana, nie możemy przyjąć, że w programie wielowątkowym obiekt `Pair` będzie się znajdować w prawidłowym stanie.

Wyobraź sobie, że ktoś oddaje Ci do dyspozycji klasę `Pair`, którą musisz zastosować w środowisku wielowątkowym, a która nie jest do tego przystosowana. Należałoby wtedy utworzyć osłonę w postaci klasy `PairManager`, która przechowuje obiekt `Pair` i zarządza dostępem do niego. Należy zauważyć, że klasa ta ma tylko dwie metody publiczne: synchronizowaną metodę `getPair()` oraz abstrakcyjną metodę `increment()`. Synchronizacja tej ostatniej metody zostanie zrealizowana w momencie jej implementacji.

W nomenklaturze wzorców projektowych[15] struktura klasy `PairManager`, której niektóre możliwości funkcjonalne są zaimplementowane w klasie bazowej posiadającej jedną lub więcej metod abstrakcyjnych zdefiniowanych w klasie pochodnej, określana jest mianem metody szablonowej (ang. *Template Method*). Wzorce projektowe pozwalają hermetyzować zmiany w kodzie; w naszym przypadku modyfikowanym fragmentem jest metoda `increment()`. W klasie `PairManager1` cała metoda `increment()` jest synchronizowana, lecz w klasie `PairManager2` synchronizowany jest wyłącznie fragment kodu tej metody. Należy pamiętać, że słowo kluczowe `synchronized` nie należy do sygnatury metody, dzięki czemu można je dodawać podczas przesłaniania metod.

Metoda `store()` dodaje obiekt klasy `Pair` do synchronizowanej kolekcji `ArrayList`, przez co ta operacja jest zabezpieczona przed wpływem wielowątkowości. Nie musi być dodatkowo chroniona, dlatego występuje poza blokiem synchronizowanym klasy `PairManager2`.

[15] Patrz książka *Design Patterns* autorstwa Gammy i innych, (Addison-Wesley, 1995).

Klasa PairManipulator służy do testowania dwóch różnych podtypów PairManager przez wywołanie metody increment() w jednym zadaniu, podczas gdy w drugim zadaniu działa klasa PairChecker. Aby sprawdzić częstotliwość wykonywania testu w klasie PairChecker, osadziliśmy licznik checkCounter zwiększany po każdym pomyślnym teście. W metodzie main() tworzone są dwa obiekty PairManager, oba uruchamiane na pewien czas, po którym wypisywane są wyniki testu.

Choć różne uruchomienia programu mogą dać diametralnie różne wyniki, zasadniczo powinno być widoczne, że metoda PairManager1.increment() nie jest dla PairChecker dostępna tak często jak PairManager2.increment(), która jest synchronizowana jedynie częściowo i przez to zawiera więcej kodu nieblokowanego. I właśnie częstotliwość dostępu jest powodem zastępowania całościowej synchronizacji metod blokami synchronizacji.

Sekcje krytyczne można też tworzyć przez jawne zastosowanie blokady w postaci obiektu Lock:

```
//: concurrency/ExplicitCriticalSection.java
// Jawny obiekt blokady tworzący sekcję krytyczną.
package concurrency;
import java.util.concurrent.locks.*;

// Synchronizowanie całej metody:
class ExplicitPairManager1 extends PairManager {
  private Lock lock = new ReentrantLock();
  public synchronized void increment() {
    lock.lock();
    try {
      p.incrementX();
      p.incrementY();
      store(getPair());
    } finally {
      lock.unlock();
    }
  }
}

// Wydzielenie sekcji krytycznej:
class ExplicitPairManager2 extends PairManager {
  private Lock lock = new ReentrantLock();
  public void increment() {
    Pair temp;
    lock.lock();
    try {
      p.incrementX();
      p.incrementY();
      temp = getPair();
    } finally {
      lock.unlock();
    }
    store(temp);
  }
}

public class ExplicitCriticalSection {
```

```
  public static void main(String[] args) throws Exception {
    PairManager
      pman1 = new ExplicitPairManager1(),
      pman2 = new ExplicitPairManager2();
    CriticalSection.testApproaches(pman1, pman2);
  }
} /* Output: (Sample)
pm1: Para: x: 15, y: 15 licznik = 174035
pm2: Para: x: 16, y: 16 licznik = 2608588
*///:~
```

Wykorzystujemy tu znaczną część kodu *CriticalSection.java* i tworzymy dodatkowy podtyp `PairManager` wykorzystujący blokadę za pomocą obiektu `Lock`. Sposób tworzenia sekcji krytycznej na bazie blokady jawnej ilustruje klasa `ExplicitPairManager`; wywołanie `store()` znajduje się w niej poza sekcją krytyczną.

Synchronizacja dostępu na bazie innych obiektów

W bloku `synchronized` należy określić obiekt, na którego podstawie blok będzie synchronizowany. Zazwyczaj najbardziej sensownym obiektem, jaki można w tej sytuacji wykorzystać, jest obiekt, którego metoda została wywołana — `synchronize(this)`. To rozwiązanie wykorzystuje klasa `PairManager2`. W takim przypadku, po uzyskaniu blokady synchronizowanego bloku kodu, inne synchronizowane metody tego samego obiektu nie mogą być wywoływane, nie dojdzie też do wykonania kodu w pozostałych blokach synchronizowanych obiektu. A zatem uzyskujemy efekt redukcji zasięgu synchronizacji.

Tymczasem czasami trzeba oprzeć synchronizację na obiekcie innym niż `this`; trzeba wtedy zadbać o to, aby wszystkie zadania dokonywały synchronizacji według tego samego obiektu. Poniższy przykład pokazuje, że dwa wątki mogą „wejść" do tego samego obiektu, jeśli dostęp do jego metod synchronizują dwie odrębne blokady.

```
//: concurrency/SyncObject.java
// Synchronizacja na bazie różnych obiektów.
import static net.mindview.util.Print.*;

class DualSynch {
  private Object syncObject = new Object();
  public synchronized void f() {
    for(int i = 0; i < 5; i++) {
      print("f()");
      Thread.yield();
    }
  }
  public void g() {
    synchronized(syncObject) {
      for(int i = 0; i < 5; i++) {
        print("g()");
        Thread.yield();
      }
    }
  }
}

public class SyncObject {
```

```
  public static void main(String[] args) {
    final DualSynch ds = new DualSynch();
    new Thread() {
      public void run() {
        ds.f();
      }
    }.start();
    ds.g();
  }
} /* Output: (Sample)
g()
f()
g()
f()
g()
f()
g()
f()
g()
f()
*///:~
```

Metoda `f()` klasy `DualSync` jest synchronizowana w oparciu o `this` (przy czym synchronizowana jest cała metoda), natomiast metoda `g()` posiada blok synchronizowany w oparciu o obiekt `syncObject`. A zatem czynniki synchronizujące dostęp do obu tych metod są różne. Udowadnia to metoda `main()`, w której tworzony jest wątek wywołujący metodę `f()`. Sama metoda `main()` wywołuje następnie metodę `g()`. Wyniki programu pokazują, że obie metody działają równocześnie, a zatem żadna z nich nie została zablokowana przez mechanizm synchronizacji drugiej.

Ćwiczenie 15. Utwórz klasę z trzema metodami zawierającymi sekcje krytyczne, wszystkie synchronizowane w oparciu o ten sam obiekt. Utwórz wiele zadań i pokaż, że w danym momencie można wykonywać tylko jedną z tych metod. Zmodyfikuj potem metody tak, aby każda z nich opierała synchronizację na innym obiekcie, i pokaż, że w takim układzie może dojść do współbieżnego wykonania wszystkich trzech metod (1).

Ćwiczenie 16. Zmodyfikuj ćwiczenie 15. z użyciem jawnego obiektu blokady (`Lock`) (1).

Lokalna pamięć wątku

Drugi sposób zapobiegania kolizjom wątków w odwołaniach do wspólnych zasobów to eliminowanie owych wspólnych zasobów. Pomocny w tym jest mechanizm *lokalnej pamięci wątku*, który automatycznie tworzy różne kopie tej samej zmiennej dla każdego z wątków korzystających z danego obiektu. Jeśli więc mamy pięć wątków korzystających z obiektu ze zmienną `x`, mechanizm lokalnej pamięci wątków może wygenerować pięć oddzielnych kopii `x` przechowywanych w odrębnych obszarach pamięci. Zmienne takie pozwalają na przypisywanie wątkom stanów.

Tworzenie pamięci lokalnej wątku i zarządzanie nią jest realizowane przez klasę `java.lang.ThreadLocal`, co widać poniżej:

```
//: concurrency/ThreadLocalVariableHolder.java
// Automatyczne wyposażenie wątków w ich własną pamięć.
import java.util.concurrent.*;
import java.util.*;
```

```
class Accessor implements Runnable {
  private final int id;
  public Accessor(int idn) { id = idn; }
  public void run() {
    while(!Thread.currentThread().isInterrupted()) {
      ThreadLocalVariableHolder.increment();
      System.out.println(this);
      Thread.yield();
    }
  }
  public String toString() {
    return "#" + id + ": " +
      ThreadLocalVariableHolder.get();
  }
}

public class ThreadLocalVariableHolder {
  private static ThreadLocal<Integer> value =
    new ThreadLocal<Integer>() {
      private Random rand = new Random(47);
      protected synchronized Integer initialValue() {
        return rand.nextInt(10000);
      }
    };
  public static void increment() {
    value.set(value.get() + 1);
  }
  public static int get() { return value.get(); }
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new Accessor(i));
    TimeUnit.SECONDS.sleep(3);  // Działanie przez pewien czas
    exec.shutdownNow();         // Wymuszenie zakończenia wszystkich
                                // obiektów Accessor
  }
} /* Output: (Sample)
#0: 9259
#1: 556
#2: 6694
#3: 1862
#4: 962
#0: 9260
#1: 557
#2: 6695
#3: 1863
#4: 963
...
*///:~
```

Obiekty klasy `ThreadLocal` są zwykle przechowywane jako pola statyczne. Do zawartości utworzonego obiektu `ThreadLocal` możemy się odwoływać jedynie za pośrednictwem metod `get()` i `set()`. Metoda `get()` zwraca kopię obiektu odpowiednią dla danego wątku, a `set()` wstawia argument wywołania do obiektu odpowiedniego dla wątku, zwracając obiekt przechowywany tu poprzednio. Ich działanie demonstrują w programie `ThreadLocalVariableHolder` metody `get()` i `increment()`. Zauważ, że żadna z nich nie jest synchronizowana, bo o eliminację sytuacji hazardowych dba sam obiekt `ThreadLocal`.

Po uruchomieniu programu przekonasz się, że każdy z wątków otrzymał do dyspozycji własną pamięć, bo każdy z nich operuje własnym licznikiem — mimo że w całym programie występuje przecież tylko jeden obiekt `ThreadLocalVariableHolder`.

Przerywanie wykonania zadań

W niektórych z prezentowanych dotąd przykładów występowały metody `cancel()` i `isCanceled()` zebrane w klasie, do której dostęp miały wszystkie zadania. Zadanie sprawdzało wywołaniem `isCanceled()`, czy ma się zakończyć. To jak najbardziej sensowne podejście do problemu przerywania działania zadań. Ale w niektórych sytuacjach zadanie trzeba przerwać bardziej zdecydowanie. Zajmiemy się więc kwestiami i problemami takiego wymuszania przerywania.

Przyjrzyjmy się na początku przykładowi ilustrującemu sedno problemu, a przy okazji stanowiącemu dodatkowy przykład współdzielenia zasobów.

Ogród botaniczny (symulacja)

W tej symulacji rada zarządzająca ogrodem zapragnęła poznać liczbę osób przechodzących codziennie przez jego bramy. Każda brama posiada przepust obrotowy albo inny rodzaj licznika, a po zwiększeniu licznika bramki dochodzi do zwiększenia licznika wspólnego dla wszystkich bramek, reprezentującego łączną liczbę odwiedzających.

```
//: concurrency/OrnamentalGarden.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Count {
  private int count = 0;
  private Random rand = new Random(47);
  // Usunięcie słowa synchronized spowoduje błąd:
  public synchronized int increment() {
    int temp = count;
    if(rand.nextBoolean()) // Oddanie (mniej więcej) połowy czasu zadania
      Thread.yield();
    return (count = ++temp);
  }
  public synchronized int value() { return count; }
}

class Entrance implements Runnable {
  private static Count count = new Count();
  private static List<Entrance> entrances =
    new ArrayList<Entrance>();
  private int number = 0;
  // Odczyt nie wymaga synchronizacji:
  private final int id;
  private static volatile boolean canceled = false;
  // Operacja atomowa na polu ulotnym:
  public static void cancel() { canceled = true; }
  public Entrance(int id) {
```

```
    this.id = id;
    // Zachowanie zadania na liście. Zapobiega usuwaniu
    // zadań w ramach odśmiecania:
    entrances.add(this);
  }
  public void run() {
    while(!canceled) {
      synchronized(this) {
        ++number;
      }
      print(this + " Razem: " + count.increment());
      try {
        TimeUnit.MILLISECONDS.sleep(100);
      } catch(InterruptedException e) {
        print("przerwano uśpienie");
      }
    }
    print("Zatrzymywanie: " + this);
  }
  public synchronized int getValue() { return number; }
  public String toString() {
    return "Wejście " + id + ": " + getValue();
  }
  public static int getTotalCount() {
    return count.value();
  }
  public static int sumEntrances() {
    int sum = 0;
    for(Entrance entrance : entrances)
      sum += entrance.getValue();
    return sum;
  }
}

public class OrnamentalGarden {
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new Entrance(i));
    // Odczekaj, zatrzymaj zadania i zbierz dane:
    TimeUnit.SECONDS.sleep(3);
    Entrance.cancel();
    exec.shutdown();
    if(!exec.awaitTermination(250, TimeUnit.MILLISECONDS))
      print("Niektóre zadania wciąż działają!");
    print("Razem: " + Entrance.getTotalCount());
    print("Suma wejść: " + Entrance.sumEntrances());
  }
} /* Output: (Sample)
Wejście 0: 1 Razem: 1
Wejście 2: 1 Razem: 3
Wejście 1: 1 Razem: 2
Wejście 4: 1 Razem: 5
Wejście 3: 1 Razem: 4
Wejście 2: 2 Razem: 6
Wejście 4: 2 Razem: 7
Wejście 0: 2 Razem: 8
...
```

```
Wejście 3: 29 Razem: 143
Wejście 0: 29 Razem: 144
Wejście 4: 29 Razem: 145
Wejście 2: 30 Razem: 147
Wejście 1: 30 Razem: 146
Wejście 0: 30 Razem: 149
Wejście 3: 30 Razem: 148
Wejście 4: 30 Razem: 150
Zatrzymywanie: Wejście 2: 30
Zatrzymywanie: Wejście 1: 30
Zatrzymywanie: Wejście 0: 30
Zatrzymywanie: Wejście 3: 30
Zatrzymywanie: Wejście 4: 30
Razem: 150
Suma wejść: 150
*///:~
```

Pojedynczy obiekt `Count` przechowuje główny licznik odwiedzających ogród; obiekt ten jest przechowywany w postaci pola statycznego w klasie zadania (`Entrance`). Metody `Count.increment()` i `Count.value()` są metodami synchronizowanymi, co zabezpiecza dostęp do pola licznika — `count`. Metoda `increment()` klasy `Count` wykorzystuje obiekt `Random` do oddawania czasu wykonania innym zadaniom w średnio co drugim wywołaniu, co odbywa się pomiędzy przepisaniem wartości licznika `count` do lokalnej zmiennej `temp`, jej zwiększeniem i zapisaniem z powrotem w liczniku `count`. Jeśli w sygnaturze metody `increment()` zrezygnujemy ze słowa kluczowego `synchronized`, program szybko się załamie, bo dojdzie do równoczesnego modyfikowania licznika `count` z poziomu wielu zadań (a wywołania `yield()` w metodzie `increment()` jeszcze przyspieszą wystąpienie problemu).

Każde zadanie `Entrance` przechowuje lokalną wartość `number` reprezentującą liczbę odwiedzających, którzy przechodzili przez daną bramkę. Obecność lokalnych liczników pozwala na dodatkowe skontrolowanie poprawności zliczania odwiedzających w liczniku głównym. Metoda `Entrance.run()` zwiększa wartość lokalnego licznika oraz licznika obiektu globalnego i usypia zadanie na 100 milisekund.

Ponieważ pole `Entrance.canceled` jest ulotnym (`volatile`) znacznikiem typu `boolean`, angażowanym wyłącznie w operacjach przypisania i odczytu (i nigdy nie jest odczytywane w połączeniu z innymi polami), można zaniechać dodatkowej synchronizacji dostępu do tego pola, zaś w razie jakichkolwiek wątpliwości co do takiego postępowania, zawsze można skorzystać z synchronizacji.

Program podejmuje dodatkowy wysiłek związany z łagodnym i porządnym przerwaniem wszystkich zadań. Ma to po pierwsze ilustrować konieczność zachowania ostrożności przy kończeniu programu wielowątkowego, a po drugie — demonstrować wartość metody `interrupt()`, o której powiemy sobie za chwilę.

Po upływie 3 sekund metoda `main()` wysyła do `Entrance` statyczny komunikat `cancel()`, po czym na rzecz obiektu wykonawcy `exec` wywołuje metodę `shutdown()`, a następnie (dla tego samego obiektu) metodę `awaitTermination()`. Metoda `ExecutorService.awaitTermination()` oczekuje na zakończenie każdego z zadań, a jeśli wszystkie zakończą się przed upływem zadanego czasu, zwraca wartość `true`; jeśli w czasie oczekiwania nie dojdzie do zakończenia wszystkich zadań, otrzymamy wartość `false`. Choć powoduje to

zakończenie przez zadania wykonywania ich metody `run()`, same obiekty `Entrance` pozostają dostępne, bo w konstruktorze każdy z nich został umieszczony w statycznym kontenerze `List<Entrance>` o nazwie `entrances`. Dlatego można śmiało wywołać statyczną metodę `sumEntrances()` sumującą wartości lokalnych liczników poszczególnych bramek.

W czasie wykonania programu na wyjściu wypisywane są numery bramek wraz z liczbą odwiedzających zarejestrowanych w danej bramce i łączną liczbą odwiedzających przebywających w ogrodzie. Jeśli z deklaracji metody `Count.increment()` usuniesz słowo `synchronized`, szybko zauważysz rozbieżność pomiędzy sumą odwiedzających w poszczególnych bramkach a wartością licznika ogólnego `count`. Dopóki dostęp do obiektu `Count` synchronizuje muteks, zliczanie działa jak najbardziej poprawnie. Pamiętaj, że metoda `Count.increment()` potęguje ryzyko błędu przez kopiowanie licznika do zmiennej lokalnej i wywoływanie `yield()`. W praktycznych zastosowaniach wielowątkowości zdarza się, że ryzyko kolizji jest statystycznie znikome, co bywa zwodnicze — mimo testów wydaje się, że wszystko działa poprawnie, a program jest najzupełniej bezpieczny. Ale, jak w omawianym przykładzie, może być tak, że problem, choć nieujawniony, będzie obecny. Przy przeglądaniu i kontrolowaniu kodu współbieżnego trzeba więc zachować najwyższą staranność i uwagę.

Ćwiczenie 17. Zaimplementuj licznik promieniowania, który będzie zbierał dane z dowolnej liczby zdalnych czujników (2).

Przerywanie zablokowanego wątku

Metoda `Entrance.run()` z poprzedniego przykładu zawiera w swojej pętli wykonania wywołanie metody `sleep()`. Wiadomo, że po wywołaniu `sleep()` zadanie zostanie w końcu wznowione i będzie miało okazję opuścić pętlę po sprawdzeniu znacznika `canceled`. Ale `sleep()` to tylko jeden z przypadków zawieszenia wykonania zadania; niekiedy zaś zadanie zawieszone trzeba przerwać, nie doczekawszy jego wznowienia.

Stany wątków

Wątek może się znajdować w jednym z czterech stanów:

1. *Nowy*. Wątek pozostaje w tym stanie jedynie chwilowo, w trakcie tworzenia. Wtedy następuje przydział niezbędnych zasobów systemowych i inicjalizacja wątku. W momencie, w którym wątek uzyska gotowość do otrzymania czasu procesora, planista zmieni oznaczenie stanu wątku na *gotowy* lub *zablokowany*.
2. *Gotowy* (ang. *runnable*). Oznacza, że wątek *może* działać, kiedy tylko mechanizm podziału czasu udostępni mu procesor. Nic nie powstrzymuje wątku, więc może być lub też nie być uruchomiony, o ile tylko zarządca może zorganizować mu dostęp do procesora; nie jest to ani stan śmierci, ani blokady.
3. *Zablokowany* (albo *zawieszony*). Wątek mógłby działać, ale jest coś, co mu przeszkadza. Gdy wątek jest w stanie zablokowania, zarządca podziału czasu po prostu będzie go pomijał i nie da mu żadnego dostępu do procesora. Dopóki wątek ponownie nie wejdzie w stan gotowości, nic będzie mógł wykonać żadnej operacji.

4. *Uśmiercony* (albo *terminalny*). Wątek, który nie otrzymuje czasu procesora i nie jest już uwzględniany przez planistę przy jego rozdziale. Jego zadanie zakończyło się i wątek nie jest już w stanie gotowości. Normalny sposób „zejścia" wątku to zakończenie jego metody `run()`, ale wątek może być też przerwany brutalniej, o czym za chwilę.

Stan zawieszenia

Zadanie może zostać zablokowane z następujących przyczyn:

- Zostało uśpione poprzez wywołanie metody `sleep(milisekundy)` i nie będzie działało przez podany czas.
- Wykonywanie zadania zostało zawieszone poprzez wywołanie metody `wait()`. Zadanie nie będzie ponownie gotowe do działania, aż do chwili gdy otrzyma komunikat `notify()` lub `notifyAll()` (albo odpowiadające im `signal()` i `signalAll()` biblioteki `java.util.concurrent` z wydania Java SE5). Zagadnieniem tym zajmiemy się w dalszej części rozdziału.
- Zadanie oczekuje na zakończenie jakiejś operacji wejścia-wyjścia.
- Zadanie usiłuje wywołać metodę synchronizowaną na innym obiekcie, a blokada tego obiektu została założona i jest przetrzymywana przez inne zadanie.

W starszym kodzie można się także spotkać z wywołaniami metody `suspend()` oraz `resume()`, służącymi odpowiednio do blokowania i odblokowywania wątków. Jednak we współczesnej Javie nie są one zalecane (gdyż zwiększają prawdopodobieństwo wystąpienia wzajemnej blokady) i dlatego nie przedstawię ich w tej książce. Zarzucono też stosowanie metody `stop()`, bo nie zwalnia ona blokad, które wątek posiada, i jeżeli obiekty są w stanie niespójnym („uszkodzone"), inne zadania mogą je w takim stanie podglądać i modyfikować. Wynikające z tego problemy mogą być subtelne i trudne do wykrycia.

Stoimy teraz przed takim oto problemem: niejednokrotnie chcemy zakończyć zadanie znajdujące się w stanie zawieszenia (zablokowane) i nie możemy pozwolić sobie na oczekiwanie na wznowienie wykonania zadania i dobrnięcie do momentu, w którym zadanie samo zorientuje się, że powinno zakończyć swoje wykonanie — musimy to wymusić.

Wymuszanie przerwania wykonania

Łatwo sobie wyobrazić, że wymuszenie wyjścia ze środka metody `run()` to sposób mocno ryzykowny w porównaniu z oczekiwaniem na sprawdzenie przez zadanie znacznika odwołania albo oczekiwaniem na dotarcie do innego miejsca, w którym programista zdecydował się na zakończenie metody. Wymuszenie nagłego przerwania wykonania oznacza niemożność wykonania operacji porządkowych. Przerywanie wykonania metody `run()` jest dla zadania czymś porównywalnym do zgłoszenia wyjątku; to najbliższa analogia i w Javie do wymuszania przerywania zadań służą właśnie wyjątki[16]

[16] Jednakże wyjątki nigdy nie są rozprowadzane asynchronicznie, nie ma więc ryzyka, że coś przerwie w pół wykonywanie instrukcji czy wywoływanie metody. I dopóki przy używaniu muteksów będziesz się trzymał idiomu `try-finally`, muteksy te będą w przypadku wyjątku automatycznie zwalniane.

(bardzo to bliskie nadużyciu wyjątków, bo bynajmniej nie są one przeznaczone do ingerowania w przepływ sterowania w programie). Aby po takim przerwaniu zadania powrócić do znanego stabilnego stanu, trzeba dokładnie prześledzić ścieżki wykonania w kodzie i odpowiednio zastosować klauzule catch z kodem porządkującym stan zadania.

Aby można było przerywać wykonanie zadań zawieszonych, klasa Thread udostępnia metodę interrupt(). Jej wywołanie ustawia dla wątku status przerwania. Wątek z ustawionym statusem przerwania, jeśli jest zablokowany albo podejmuje próbę wykonania operacji blokującej, zgłosi wyjątek InterruptedException. Po zgłoszeniu wyjątku albo wywołaniu przez zadanie metody Thread.interrupted() status przerwania zostanie wyzerowany. Metoda Thread.interrupted() jest więc drugim sposobem opuszczenia pętli metody run(), i to bez zgłaszania wyjątku.

Aby wywołać metodę interrupt(), trzeba dysponować obiektem klasy Thread. Zauważyłeś już zapewne, że implementacja biblioteki współbieżności unika bezpośredniego manipulowania egzemplarzami klasy Thread, zakładając pośrednictwo wykonawców (obiektów Executor). Otóż wywołanie metody shutdownNow() na rzecz obiektu wykonawcy jest równoznaczne z wywołaniem metody interrupt() dla każdego z wątków uruchomionych i zarządzanych przez tego wykonawcę. To wygodne, bo zazwyczaj chcemy zatrzymać wszystkie zadania danego wykonawcy za jednym zamachem. Ale bywa i tak, że chcemy przerwać tylko jedno, konkretne zadanie. Wykorzystując obiekty wykonawców, możemy zachować dostęp do kontekstu zadań, jeśli uruchomimy je nie wywołaniem execute(), a metodą submit(). Metoda submit() zwraca referencję Future<?> z niedookreślonym parametrem typowym, bo na jej rzecz nigdy nie wykonuje się wywołań metody get() — celem udostępniania takiej referencji jest umożliwienie wywołania cancel() i tym samym przerwania wykonania danego zadania. Jeśli do metody cancel() przekazana zostanie wartość true, będzie ona oznaczać przyzwolenie na wywołanie interrupt() na rzecz danego wątku w celu jego zatrzymania; cancel() stanowi więc środek przerywania wykonania pojedynczych zadań uruchamianych za pośrednictwem wykonawcy.

Oto podstawy stosowania metody interrupt() w połączeniu z obiektami-wykonawcami:

```
// : concurrency/Interrupting.java
// Przerywanie zablokowanego wątku.
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class SleepBlocked implements Runnable {
  public void run() {
    try {
      TimeUnit.SECONDS.sleep(100);
    } catch(InterruptedException e) {
      print("InterruptedException");
    }
    print("Opuszczanie metody SleepBlocked.run()");
  }
}

class IOBlocked implements Runnable {
  private InputStream in;
  public IOBlocked(InputStream is) { in = is; }
```

```
    public void run() {
      try {
        print("Oczekiwanie na read():");
        in.read();
      } catch(IOException e) {
        if(Thread.currentThread().isInterrupted()) {
          print("Przerwany w zawieszeniu na operacji wejścia-wyjścia");
        } else {
          throw new RuntimeException(e);
        }
      }
      print("Opuszczanie metody IOBlocked.run()");
    }
  }

  class SynchronizedBlocked implements Runnable {
    public synchronized void f() {
      while(true) // Nigdy nie zwalnia blokady
        Thread.yield();
    }
    public SynchronizedBlocked() {
      new Thread() {
        public void run() {
          f(); // Blokada pozyskana przez ten wątek
        }
      }.start();
    }
    public void run() {
      print("Próba wywołania f()");
      f();
      print("Opuszczanie metody SynchronizedBlocked.run()");
    }
  }

  public class Interrupting {
    private static ExecutorService exec =
      Executors.newCachedThreadPool();
    static void test(Runnable r) throws InterruptedException{
      Future<?> f = exec.submit(r);
      TimeUnit.MILLISECONDS.sleep(100);
      print("Przerywanie " + r.getClass().getName());
      f.cancel(true); // Przerwanie, jeśli działa
      print("Nakaz przerwania wysłany do " + r.getClass().getName());
    }
    public static void main(String[] args) throws Exception {
      test(new SleepBlocked());
      test(new IOBlocked(System.in));
      test(new SynchronizedBlocked());
      TimeUnit.SECONDS.sleep(3);
      print("Opuszczanie programu (System.exit(0))");
      System.exit(0); // ... dwa ostatnie przerwania zawiodły
    }
  } /* Output: (95% match)
  Przerywanie SleepBlocked
  InterruptedException
  Opuszczanie metody SleepBlocked.run()
  Nakaz przerwania wysłany do SleepBlocked
  Oczekiwanie na read():
```

```
Przerywanie IOBlocked
Nakaz przerwania wysłany do IOBlocked
Próba wywołania f()
Przerywanie SynchronizedBlocked
Nakaz przerwania wysłany do SynchronizedBlocked
Opuszczanie programu (System.exit(0))
*///:~
```

Każde z zadań programu reprezentuje inny rodzaj blokowania. `SleepBlocked` to przykład blokowania przerywalnego, podczas gdy `IOBlocked` i `SynchronizedBlocked` to blokowanie niedające się przerywać[17]. Program pokazuje, że zawieszenie w oczekiwaniu na operację wejścia-wyjściu oraz zawieszenie w oczekiwaniu na zwolnienie blokady synchronizacyjnej nie dają się przerywać; widać to zresztą również w samym kodzie: przy operacjach wejścia-wyjścia i próbach wywołania metod synchronizowanych nie widać bloków obsługi wyjątku `InterruptedException`.

Pierwsze dwa przypadki są zupełnie oczywiste: metoda `run()` wywołuje w pierwszej klasie metodę `sleep()`, a w drugiej — metodę `read()`. Aby jednak zademonstrować zachowanie zawieszenia na blokadzie synchronizującej (`SynchronizedBlocked`), musimy najpierw pozyskać blokadę. Odbywa się to w konstruktorze tworzącym egzemplarz nienazwanej klasy `Thread` zakładający blokadę w wyniku wywołania metody `f()` (musi to być inny wątek niż wątek metody `run()` zadania `SynchronizedBlocked`, bo ta sama blokada może być w ramach tego samego zadania pozyskiwana wielokrotnie). Ponieważ metoda `f()` nigdy nie zwraca sterowania do wywołującego, nie doczekamy się zwolnienia blokady. Zadanie `SynchronizedBlocked` w metodzie `run()` próbuje wywołać metodę `f()` i zostaje zawieszone w oczekiwaniu na zwolnienie blokady.

Na wyjściu programu widać, że udało się przerwać wywołanie `sleep()` (dotyczy to każdego wywołania wymagającego przechwytywania wyjątku `InterruptedException`). Ale nie można przerwać zadania, które oczekuje na zwolnienie blokady albo na zakończenie operacji wejścia-wyjścia. To nieco niepokojące, zwłaszcza przy tworzeniu zadań specjalizowanych do obsługi operacji wejścia-wyjścia, bo oznacza, że te operacje mogą potencjalnie zablokować program wielowątkowy. To zmartwienie trapi zwłaszcza programistów aplikacji opartych na WWW.

Nieco toporne ale skuteczne rozwiązanie problemu polega likwidowaniu przyczyny blokady, czyli na zamknięciu zasobu, na którym zablokowało się zadanie:

```
//: concurrency/CloseResource.java
// Przerywanie zablokowanego zadania
// przez zamykanie zasobu.
// {RunByHand}
import java.net.*;
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;
```

[17] W niektórych wydaniach JDK zawarta była obsługa wyjątku `InterruptedIOException`, ale była ona zaimplementowana jedynie częściowo i tylko dla niektórych platform. Zgłoszenie takiego wyjątku sprawiało, że obiekty wejścia-wyjścia traciły wszelką użyteczność. W przyszłych wydaniach Javy nie należy więc oczekiwać dalszej obecności tego wyjątku.

```
public class CloseResource {
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    ServerSocket server = new ServerSocket(8080);
    InputStream socketInput =
      new Socket("localhost", 8080).getInputStream();
    exec.execute(new IOBlocked(socketInput));
    exec.execute(new IOBlocked(System.in));
    TimeUnit.MILLISECONDS.sleep(100);
    print("Zamykanie wszystkich wątków");
    exec.shutdownNow();
    TimeUnit.SECONDS.sleep(1);
    print("Zamykanie " + socketInput.getClass().getName());
    socketInput.close(); // Zwolnienie zablokowanego wątku
    TimeUnit.SECONDS.sleep(1);
    print("Zamykanie " + System.in.getClass().getName());
    System.in.close(); // Zwolnienie zablokowanego wątku
  }
} /* Output: (85% match)
Oczekiwanie na read():
Oczekiwanie na read():
Zamykanie wszystkich wątków
Zamykanie java.net.SocketInputStream
Przerwany w zawieszeniu na operacji wejścia-wyjścia
Opuszczanie metody IOBlocked.run()
Zamykanie java.io.BufferedInputStream
Opuszczanie metody IOBlocked.run()
*///:~
```

Po wywołaniu `shutdownNow()` wprowadzamy opóźnienie pomiędzy wywołaniem `close()` na rzecz dwóch strumieni wejściowych uwypuklając fakt zamykania zasobu, na którym zablokowało się zadanie. Co ciekawe, wywołanie `interrupt()` pojawia się przy zamykaniu gniazda `Socket`, ale już nie przy zamykaniu strumienia `System.in`.

Na szczęście klasy `nio`, zaprezentowane w rozdziale „Wejście-wyjście", pozwalają na bardziej cywilizowane przerwanie operacji. Zablokowane kanały `nio` reagują na przerwanie automatycznie:

```
//: concurrency/NIOInterruption.java
// Przerywanie zablokowanego kanału NIO.
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class NIOBlocked implements Runnable {
  private final SocketChannel sc;
  public NIOBlocked(SocketChannel sc) { this.sc = sc; }
  public void run() {
    try {
      print("Oczekiwanie na read() w " + this);
      sc.read(ByteBuffer.allocate(1));
    } catch(ClosedByInterruptException e) {
      print("ClosedByInterruptException");
    } catch(AsynchronousCloseException e) {
      print("AsynchronousCloseException");
```

```
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
      print("Opuszczanie metody NIOBlocked.run() " + this);
    }
  }

  public class NIOInterruption {
    public static void main(String[] args) throws Exception {
      ExecutorService exec = Executors.newCachedThreadPool();
      ServerSocket server = new ServerSocket(8080);
      InetSocketAddress isa =
        new InetSocketAddress("localhost", 8080);
      SocketChannel sc1 = SocketChannel.open(isa);
      SocketChannel sc2 = SocketChannel.open(isa);
      Future<?> f = exec.submit(new NIOBlocked(sc1));
      exec.execute(new NIOBlocked(sc2));
      exec.shutdown();
      TimeUnit.SECONDS.sleep(1);
      // Przerwanie za pośrednictwem wywołania cancel:
      f.cancel(true);
      TimeUnit.SECONDS.sleep(1);
      // Zwolnienie blokady przez zamknięcie kanału:
      sc2.close();
    }
  } /* Output: (Sample)
  Oczekiwanie na read() w NIOBlocked@7a84e4
  Oczekiwanie na read() w NIOBlocked@15c7850
  ClosedByInterruptException
  Opuszczanie metody NIOBlocked.run() NIOBlocked@15c7850
  AsynchronousCloseException
  Opuszczanie metody NIOBlocked.run() NIOBlocked@7a84e4
  *///:~
```

Jak widać, można też zamknąć kanał, aby zwolnić blokadę, choć rzadko powinno to być konieczne. Zauważ, że wykorzystywanie `execute()` do uruchamiania obu zadań i wywoływanie `e.shutdownNow()` pozwala na proste przerwanie wszystkiego; przechwytywanie obiektu `Future` w powyższym przykładzie było potrzebne jedynie do selektywnego przesłania przerwania do jednego, konkretnego wątku[18].

Ćwiczenie 18. Utwórz klasę (niebędącą zadaniem) z metodą wywołującą `sleep()` na dość długi czas. Utwórz zadanie, które wywoła metodę z klasy niebędącej zadaniem. W metodzie `main()` uruchom to zadanie, a następnie wywołaj metodę `interrupt()` w celu przerwania jego wykonania. Upewnij się, że zostało bezpiecznie zakończone (2).

Ćwiczenie 19. Zmodyfikuj program *OrnamentalGarden.java* tak, aby korzystał z metody `interrupt()` (4).

Ćwiczenie 20. Zmodyfikuj program *CachedThreadPool.java* tak, aby wszystkie zadania otrzymały komunikat `interrupt()` przed zakończeniem (1).

[18] W napisaniu tego podrozdziału pomógł mi Ervin Varga.

Blokada na muteksie

W programie *Interrupting.java* wywołanie metody synchronizowanej na rzecz obiektu, którego blokada została już założona przez inne zadanie, powoduje zawieszenie (zablokowanie) zadania wywołującego do czasu zwolnienia blokady. Poniższy przykład pokazuje, że ten sam muteks może być skutecznie wielokrotnie blokowany przez to samo zadanie:

```
//: concurrency/MultiLock.java
// Jeden wątek może wielokrotnie pozyskiwać tę samą blokadę.
import static net.mindview.util.Print.*;

public class MultiLock {
  public synchronized void f1(int count) {
    if(count-- > 0) {
      print("f1() wywołuje f2() z licznikiem " + count);
      f2(count);
    }
  }
  public synchronized void f2(int count) {
    if(count-- > 0) {
      print("f2() wywołuje f1() z licznikiem " + count);
      f1(count);
    }
  }
  public static void main(String[] args) throws Exception {
    final MultiLock multiLock = new MultiLock();
    new Thread() {
      public void run() {
        multiLock.f1(10);
      }
    }.start();
  }
} /* Output:
f1() wywołuje f2() z licznikiem 9
f2() wywołuje f1() z licznikiem 8
f1() wywołuje f2() z licznikiem 7
f2() wywołuje f1() z licznikiem 6
f1() wywołuje f2() z licznikiem 5
f2() wywołuje f1() z licznikiem 4
f1() wywołuje f2() z licznikiem 3
f2() wywołuje f1() z licznikiem 2
f1() wywołuje f2() z licznikiem 1
f2() wywołuje f1() z licznikiem 0
*///:~
```

W metodzie `main()` tworzony jest wątek (`Thread`) wywołujący metodę `f1()`, po czym następuje sekwencja wzajemnych wywołań metod `f1()` i `f2()` aż do momentu wyzerowania licznika `count`. Ponieważ zadanie już w pierwszym wywołaniu `f1()` pozyskało blokadę `multiLock`, to samo zadanie może bez problemu założyć tę samą blokadę dla potrzeb wywołania `f2()` i tak dalej. To zasadne, bowiem pojedyncze zadanie powinno mieć możliwość swobodnego wywoływania synchronizowanych metod danego obiektu; wszak zadanie to już założyło wymaganą blokadę.

Przekonaliśmy się niedawno, że za każdym razem, kiedy zadanie może zostać zablokowane w sposób niedający się przerwać, pojawia się ryzyko zawieszenia całego programu. Jednym z udogodnień implementacji bibliotek współbieżności w Javie SE5 jest możliwość blokowania zadań zablokowanych na blokadach ReentrantLock, w odróżnieniu od zadań zablokowanych na wywołaniach metod synchronizowanych i wejściach do sekcji krytycznych:

```
//: concurrency/Interrupting2.java
// Przerywanie zadania zawieszonego na blokadzie ReentrantLock.
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class BlockedMutex {
  private Lock lock = new ReentrantLock();
  public BlockedMutex() {
    // natychmiastowe założenie blokady w celu zademonstrowania
    // przerywania zadania zawieszonego na blokadzie ReentrantLock:
    lock.lock();
  }
  public void f() {
    try {
      // Blokada niedostępna dla drugiego zadania
      lock.lockInterruptibly(); // Wywołanie specjalne
      print("Założono blokadę w f()");
    } catch(InterruptedException e) {
      print("Przerwanie zakładania blokady w f()");
    }
  }
}

class Blocked2 implements Runnable {
  BlockedMutex blocked = new BlockedMutex();
  public void run() {
    print("Oczekiwanie na f() w BlockedMutex");
    blocked.f();
    print("Wytrącony z zablokowanego wywołania");
  }
}

public class Interrupting2 {
  public static void main(String[] args) throws Exception {
    Thread t = new Thread(new Blocked2());
    t.start();
    TimeUnit.SECONDS.sleep(1);
    System.out.println("Wywołanie t.interrupt()");
    t.interrupt();
  }
} /* Output:
Oczekiwanie na f() w BlockedMutex
Wywołanie t.interrupt()
Przerwanie zakładania blokady w f()
Wytrącony z zablokowanego wywołania
*///:~
```

Klasa BlockedMutex posiada konstruktor pozyskujący raz na zawsze blokadę własną obiektu. Z tego powodu próba wywołania metody f() obiektu z poziomu drugiego zadania powoduje zawieszenie tego zadania z racji niemożności założenia blokady. W klasie Blocked2 metoda run() zostanie zatrzymana na wywołaniu blocked.f(). Uruchomienie programu pokaże, że w przeciwieństwie do operacji wejścia-wyjścia, w przypadku wywołań zablokowanych na muteksach metoda interrupt() skutecznie przerywa zadanie[19].

Sprawdzanie przerwania

Zauważ, że kiedy wywołasz na rzecz wątku metodę interrupt(), do przerwania wątku dojdzie jedynie wtedy, kiedy wątek będzie na wejściu albo wewnątrz operacji blokującej (z wyjątkiem — jak się przekonałeś — nieprzerywalnych operacji wejścia-wyjścia i oczekiwania na wejście do metody synchronizowanej, kiedy po prostu nie da się przerwać wątku). Ale co, jeśli zadanie nie inicjuje wywołań blokujących albo uzależnia ich wykonywanie od jakichś zmiennych okoliczności? Czyżby poza zgłoszeniem wyjątku albo wykonaniem wywołania blokującego nie było możliwości sprowokowania zadania do opuszczenia pętli metody run()? Otóż nie, tylko że zadanie musi korzystać z dodatkowego sposobu kończenia działania, kiedy akurat nie wykonuje żadnych operacji blokujących.

Możliwość tę daje tak zwany *status przerwania*, ustawiany przez metodę interrupt(). Status przerwania sprawdza się z poziomu zadania wywołaniem metody interrupted(). Informuje ona o tym, czy dla zadania wywołano metodę interrupt(), a przy okazji zeruje status przerwania. Zerowanie to zapobiega wielokrotnemu sygnalizowaniu przerwania. Powiadomienie to jest rozprowadzane zawsze w postaci wyjątku InterruptedException albo zakończonego sukcesem wywołania Thread.interrupted(). Aby umożliwić sobie późniejsze sprawdzenie żądania przerwania, należałoby zachować gdzieś w zadaniu wynik wywołania Thread.interrupted().

Poniższy przykład ilustruje typowy idiom stosowany w metodzie run() do obsługiwania wszelkich możliwości przerywania wykonania zadania:

```
// : concurrency/InterruptingIdiom.java
// Ogólny schemat obsługi przerywania zadania.
// {Args: 1100}
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class NeedsCleanup {
  private final int id;
  public NeedsCleanup(int ident) {
    id = ident;
    print("NeedsCleanup " + id);
  }
  public void cleanup() {
    print("Porządkowanie " + id);
  }
}
```

[19] Zauważ, że choć to mało prawdopodobne, to wywołanie t.interrupt() może zostać wykonane jeszcze przed wywołaniem blocked.f().

```
class Blocked3 implements Runnable {
  private volatile double d = 0.0;
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // punkt1
        NeedsCleanup n1 = new NeedsCleanup(1);
        // Rozpoczęcie bloku try-finally zaraz za definicją
        // n1, w celu zagwarantowania właściwych porządków n1:
        try {
          print("Drzemka");
          TimeUnit.SECONDS.sleep(1);
          // punkt2
          NeedsCleanup n2 = new NeedsCleanup(2);
          // Zapewnianie właściwego porządkowania n2:
          try {
            print("Obliczenia");
            // Czasochłonna operacja nieblokująca:
            for(int i = 1; i < 2500000; i++)
              d = d + (Math.PI + Math.E) / d;
            print("Zakończono czasochłonne obliczenia");
          } finally {
            n2.cleanup();
          }
        } finally {
          n1.cleanup();
        }
      }
      print("Wyjście przewidziane w teście while()");
    } catch(InterruptedException e) {
      print("Wyjście wymuszone wyjątkiem InterruptedException");
    }
  }
}

public class InterruptingIdiom {
  public static void main(String[] args) throws Exception {
    if(args.length != 1) {
      print("Stosowanie: java InterruptingIdiom opóźnienie-w-ms");
      System.exit(1);
    }
    Thread t = new Thread(new Blocked3());
    t.start();
    TimeUnit.MILLISECONDS.sleep(new Integer(args[0]));
    t.interrupt();
  }
} /* Output: (Sample)
NeedsCleanup 1
Drzemka
NeedsCleanup 2
Obliczenia
Zakończono czasochłonne obliczenia
Porządkowanie 2
Porządkowanie 1
NeedsCleanup 1
Drzemka
Porządkowanie 1
Wyjście wymuszone wyjątkiem InterruptedException
*///:~
```

Klasa NeedsCleanup reprezentuje zasób, którego stan trzeba koniecznie uporządkować również w przypadku opuszczenia pętli wymuszonego wyjątkiem. Wszystkie zasoby NeedsCleanup tworzone w Blocked3.run() muszą być natychmiast uzupełnione blokiem try-finally, gwarantującym każdorazowe wywołanie metody porządkującej cleanup().

Program należy wywołać z argumentem określającym opóźnienie (wyrażone w milisekundach) do wywołania metody interrupt(). Używając różnych opóźnień, możesz wymuszać opuszczanie Blocked3.run() w rozmaitych miejscach pętli: w blokującym wywołaniu sleep() albo w nieblokującej części realizującej obliczenia matematyczne. Jeśli metoda interrupt() zostanie wywołana po przejściu do części oznaczonej komentarzem „punkt2" (a więc w trakcie nieblokującego obliczenia), najpierw dochodzi do zakończenia przebiegu pętli, potem do usunięcia wszystkich obiektów lokalnych, a wreszcie do opuszczenia pętli w zwykłym trybie wymuszonego warunkiem kontynuacji pętli. Jeśli jednak wywołanie interrupt() nastąpi pomiędzy punktami „punkt1" a „punkt2" (czyli za instrukcją while, ale przed albo w trakcie blokującego wywołania sleep()), zadanie jest przerywane wyjątkiem InterruptedException zaraz po tym, jak podejmuje próbę wykonania operacji blokującej. W tym przypadku porządkowane są jedynie te obiekty NeedsCleanup, które zostały utworzone do miejsca zgłoszenia wyjątku; ewentualną resztę operacji porządkowych można wykonać w klauzuli catch.

Klasa zaprojektowana pod kątem reagowania na wywołania interrupt() powinna wdrażać reguły zapewniające spójność stanu. Oznacza to zasadniczo, że tworzenie wszelkich obiektów powinno być zabezpieczane blokami try-finally tak, aby niezależnie od sposobu opuszczenia pętli metody run() zawsze dochodziło do wykonania operacji porządkujących. Taki kod może działać poprawnie, ale z racji braku automatycznych wywołań destruktorów w Javie poprawność ta jest uwarunkowana solidnością programistów-klientów, którzy są odpowiedzialni za stosowanie bloków try-finally.

Współdziałanie wątków

Przekonałeś się, że kiedy za pomocą wątków wykonujesz kilka zadań równocześnie, musisz zapobiegać niekorzystnym efektom współbieżności zadań w zakresie dostępu do wspólnych zasobów przez stosowanie muteksów (blokad) synchronizujących dostęp. Jeśli dwa zadania operują na wspólnym zasobie (zwykle jest nim pamięć), trzeba zastosować muteks, aby tylko jeden z nich mógł to robić w danym momencie.

Po rozwiązaniu problemu kolizji należałoby nauczyć się zmuszania zadań do współpracy, tak aby można było zaprzęgać wiele zadań do rozwiązywania wspólnych problemów. Nie chodzi tu o niepożądane współoddziaływanie, ale o współpracę w porozumieniu, bo zwykle części problemu nie mogą być rozwiązywane niezależnie — niektóre z nich muszą być gotowe przed rozpoczęciem pozostałych. Przypomina to planowanie harmonogramu wykonania projektu: przed zalaniem fundamentów trzeba najpierw je wykopać, ale na gotowych fundamentach można wznosić ściany niezależnie od siebie; z kolei wszystkie te zadania muszę się zakończyć przed przystąpieniem do montowania zadaszenia — i tak dalej. Niektóre z tych zadań warunkują dalsze posuwanie projektu naprzód.

Kluczowym mechanizmem umożliwiającym takie współdziałanie jest wymiana z potwierdzeniem. Do realizacji takiej wymiany skorzystamy ze znanego już środka w postaci muteksu, który tym razem będzie pilnował, aby tylko jedno zadanie odpowiadało na sygnał. Tak eliminuje się sytuacje hazardowe. Kiedy opanujemy muteksy, przyjrzymy się sposobowi zawieszania wykonania zadania do czasu zajścia jakichś okoliczności zewnętrznych (np. „wylano fundamenty"), które umożliwią podjęcie dalszego wykonania. Na pierwszy ogień pójdą kwestie związane z wymianą komunikatów z potwierdzaniem, implementowaną bezpiecznie za pośrednictwem metod `wait()` i `notifyAll()` klasy `Object`. Biblioteka współbieżności w Javie SE5 uzupełnia tę implementację klasą `Condition` z jej metodami `await()`i `signal()`. Po drodze przyjrzymy się typowym dla tego zagadnienia problemom i ich rozwiązaniom.

Metody wait() i notifyAll()

Metoda `wait()` pozwala na zawieszenie wykonania zadania w oczekiwaniu na zmianę tych warunków jego wykonania, które pozostają poza jego kontrolą. Często na oczekiwane okoliczności wpływ mają inne zadania aplikacji. Nie trzeba wtedy stosować jałowych pętli testujących spełnienie warunków dalszego wykonania — taka technika, zwana *oczekiwaniem aktywnym* (ang. *busy waiting*), to czyste marnotrawstwo cennych cykli procesora. Stąd metoda `wait()` zawieszająca zadanie w oczekiwaniu na zmianę otoczenia przerywanym wywołaniem metod `notify()` albo `notifyAll()`, które sygnalizują zajście jakichś zmian; wtedy zadanie jest „wybudzane" i może sprawdzić, czy okoliczności pozwalają na podjęcie dalszego wykonania. Metoda `wait()` stanowi więc mechanizm synchronizacji pomiędzy zadaniami.

Ważne jest, aby zrozumieć, że wywołanie metody `sleep()` *nie zwalnia* blokady obiektu, tak samo jak nie czyni tego wywołanie `yield()`. Z drugiej strony, wywołanie `wait()` zainicjowane w obrębie synchronizowanej metody wymusza zawieszenie wątku i zwolnienie blokady danego obiektu. Zwolnienie to sprawia, że inne metody synchronizowane w obiekcie wątku mogą być wywoływane podczas oczekiwania. To kwestia o zasadniczym znaczeniu, bo to właśnie owe inne metody zazwyczaj wprowadzają oczekiwane zmiany i powodują wybudzenie zawieszonego wątku. Wywołanie `wait()` oznacza więc: „zrobiłem, co mogłem, więc poczekamy tu na was, a wy możecie teraz wykonywać pozostałe operacje synchronizowane".

Istnieją dwie formy metody `wait()`. Pierwsza pobiera argument (wyrażony w milisekundach) o tym samym znaczeniu, co w przypadku metody `sleep()` — wymuszający wstrzymanie na wskazany czas. Ale w przeciwieństwie do `sleep()` wywołanie `wait(pauza)` oznacza, że:

1. Podczas wywołania metody `wait()` blokada obiektu jest zwalniana.
2. Działanie metody `wait()` może zostać zakończone w wyniku wywołania `notify()`, `notifyAll()` lub po upłynięciu podanego czasu.

Druga, powszechniej stosowana wersja metody `wait()` nie pobiera żadnych argumentów. Jej działanie sprowadza się do zawieszenia oczekiwania bez ograniczenia czasowego, do odebrania komunikatu `notify()` albo `notifyAll()`.

Jednym z unikalnych aspektów metody `wait()`, `notify()` oraz `notifyAll()` jest fakt, że w odróżnieniu od metody `sleep()` są one częściami klasy bazowej `Object`, a nie klasy `Thread`. Choć z początku może się wydawać dziwne, że mechanizm związany wyłącznie z wielowątkowością jest implementowany w uniwersalnej klasie bazowej, to jednak jest to niezwykle istotne, gdyż metody operują blokadą, która też jest częścią każdego obiektu. W rezultacie można zamieścić `wait()` w dowolnej metodzie synchronizowanej, niezależnie od tego czy klasa rozszerza `Thread` czy implementuje `Runnable`. W rzeczywistości wywołania `wait()`, `notify()` oraz `notifyAll()` można umieszczać wyłącznie w synchronizowanej metodzie lub bloku kodu (`sleep()` można wywoływać także w metodach, które nie są synchronizowane, gdyż nie manipuluje ona blokadą). Jeśli którakolwiek z tych metod zostanie wywołana w niesynchronizowanej metodzie, spowoduje to zgłoszenie wyjątku `IllegalMonitorStateException` z nieco nieintuicyjnym komunikatem „current thread not owner". Komunikat ten oznacza, że metoda wywołująca `wait()`, `notify()` lub `notifyAll()` musi „posiadać" blokadę obiektu, zanim którakolwiek z powyższych metod zostanie wywołana.

Można poprosić inny obiekt, aby wykonał operację manipulującą jego własną blokadą. W tym celu należy najpierw przechwycić blokadę tego obiektu. Na przykład, aby wywołać `notify()` obiektu `x`, należy zrobić to wewnątrz bloku `synchronized` uzyskującego blokadę obiektu `x`:

```
synchronized(x) {
  x.notify();
}
```

Spójrzmy na prosty przykład. Program *WaxOMatic.java* składa się z dwóch procesów: jeden nanosi wosk na samochód (`Car`), drugi zajmuje się polerką. Zadanie polerowania nie może podjąć wykonania przed zakończeniem woskowania, a woskowanie musi poczekać na zakończenie polerowania w celu nałożenia drugiej warstwy wosku. Oba zadania (`WaxOn` i `WaxOff`) operują na obiekcie `Car`, który używa metod `wait()` i `notifyAll()` do zawieszania i wznawiania zadań w zależności od okoliczności:

```
//: concurrency/waxomatic/WaxOMatic.java
// Prosta współpraca zadań.
package concurrency.waxomatic;
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Car {
  private boolean waxOn = false;
  public synchronized void waxed() {
    waxOn = true; // Auto gotowe do polerowania
    notifyAll();
  }
  public synchronized void buffed() {
    waxOn = false; // Auto gotowe do nałożenia
                   // następnej warstwy wosku
    notifyAll();
  }
  public synchronized void waitForWaxing()
  throws InterruptedException {
    while(waxOn == false)
      wait();
  }
```

```
  public synchronized void waitForBuffing()
  throws InterruptedException {
    while(waxOn == true)
      wait();
  }
}

class WaxOn implements Runnable {
  private Car car;
  public WaxOn(Car c) { car = c; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        printnb("Woskowanie! ");
        TimeUnit.MILLISECONDS.sleep(200);
        car.waxed();
        car.waitForBuffing();
      }
    } catch(InterruptedException e) {
      print("Wyjście wymuszone przerwaniem");
    }
    print("Koniec zadania woskowania");
  }
}

class WaxOff implements Runnable {
  private Car car;
  public WaxOff(Car c) { car = c; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        car.waitForWaxing();
        printnb("Polerowanie! ");
        TimeUnit.MILLISECONDS.sleep(200);
        car.buffed();
      }
    } catch(InterruptedException e) {
      print("Wyjście wymuszone przerwaniem");
    }
    print("Koniec zadania polerowania");
  }
}

public class WaxOMatic {
  public static void main(String[] args) throws Exception {
    Car car = new Car();
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(new WaxOff(car));
    exec.execute(new WaxOn(car));
    TimeUnit.SECONDS.sleep(5); // Niech się kręci przez jakiś czas...
    exec.shutdownNow(); // Przerwanie wszystkich zadań
  }
} /* Output: (95% match)
Woskowanie! Polerowanie! Woskowanie! Polerowanie! Woskowanie! Polerowanie! Woskowanie!
Polerowanie! Woskowanie! Polerowanie! Woskowanie! Polerowanie! Woskowanie! Polerowanie!
Woskowanie! Polerowanie! Woskowanie! Polerowanie! Woskowanie! Polerowanie! Woskowanie!
Polerowanie! Woskowanie! Polerowanie! Woskowanie!
Wyjście wymuszone przerwaniem
```

```
Koniec zadania woskowania
Wyjście wymuszone przerwaniem
Koniec zadania polerowania
*///:~
```

Klasa `Car` posiada pojedyncze pole (typu `boolean`) `WaxOn` sygnalizujące stan procesu woskowania-polerowania.

W metodzie `waitForWaxing()` odbywa się sprawdzenie stanu znacznika `waxOn`, a jeśli ma on wartość `false`, zawieszenie zadania przez wywołanie `wait()`. Odbywa się to w metodzie synchronizowanej, kiedy zadanie pozyskało blokadę obiektu. Wywołanie `wait()` powoduje zawieszenie wykonania zadania i zwolnienie blokady. Zwolnienie blokady przy zawieszaniu ma tu zasadnicze znaczenie, bo w czasie, kiedy wątek pozostaje w zawieszeniu, powinno dojść do bezpiecznej zmiany stanu obiektu (na przykład zmiana wartości `waxOn` na `true`, tak aby zadanie mogło kontynuować pracę), a to z kolei wymaga, aby jakieś inne zadanie mogło skutecznie założyć blokadę. W tym przykładzie, kiedy owo inne zadanie wywołuje metodę `waxed()` na rzecz obiektu `Car`, musi pozyskać blokadę. W ramach `waxed()` dochodzi do zmiany `waxOn` na `true` i wywołanie metody `notifyAll()`, która wybudza wszystkie zadania zawieszone wywołaniem `wait()`. Z kolei zadanie wybudzane z `wait()` musi mieć możliwość odzyskania blokady zwolnionej przy wywołaniu `wait()`. Zadanie to nie zostanie podjęte dopóty, dopóki ta blokada nie będzie wolna[20].

Metoda `WaxOn.run()` reprezentuje pierwszy etap woskowania samochodu, więc to ona zaczyna ten proces: najpierw wywołaniem `sleep()` symuluje czas niezbędny do nałożenia warstwy wosku, a następnie sygnalizuje gotowość auta do polerowania i wywołuje metodę `waitForBuffing()` zawieszającą zadanie na metodzie `wait()` do czasu, kiedy zadanie `WaxOff` wywoła metodę `buffed()` zmieniającą stan `Car` i wywołującą `notifyAll()`. Z kolei zadanie `WaxOff.run()` od razu przechodzi w stan oczekiwania wywołaniem `waitForWaxing()` i trwa w zawieszeniu do czasu nałożenia wosku przez `WaxOn` i wywołania `waxed()`. Po uruchomieniu programu można obserwować przeplatanie się obu etapów procesu woskowania przez przekazywanie sterowania pomiędzy zadaniami. Po pięciu sekundach tej krzątaniny wywołanie `interrupt()` przerywa oba wątki zadań; wywołanie `shutdownNow()` na rzecz obiektu-wykonawcy inicjuje bowiem wywołanie `interrupt()` dla wszystkich zarządzanych przez niego zadań.

Przykład ten uwypukla fakt, że wywołania metody `wait()` należy ujmować w pętli `while` sprawdzającej wystąpienie oczekiwanych okoliczności. To ważne, bo:

- Ta sama blokada może wstrzymywać kilka zadań zawieszonych z tego samego powodu; pierwsze wybudzone zadanie może zmienić sytuację (jeśli nawet nie przewidujesz tego we własnym kodzie, musisz liczyć się z programistami-klientami, dziedziczącymi po klasach zadań). W takim przypadku kolejne wybudzane zadania powinny znów zawiesić wykonanie, aż powtórnie zajdą okoliczności umożliwiające ich wznowienie.

[20] Na niektórych platformach można wybudzać zadanie z `wait()` na trzeci sposób: przez tak zwane *wybudzanie spontaniczne*. Polega ono na tym, że wątek może przedwcześnie przerwać stan zawieszenia (oczekując na zmiennej warunkowej albo semaforze) bez pomocy ze strony `notify()` czy `notifyAll()` (albo odpowiedników tych metod z klasy `Condition`). Zadanie po prostu wybudza się pozornie samo. Do takiego wybudzania dochodzi z racji zawiłości implementacji wątków POSIX na niektórych platformach. Dopuszczenie takich wybudzeń znacznie ułatwia implementację bibliotek w rodzaju pthreads.

- Kiedy zadanie jest wybudzane z `wait()`, jakieś inne zadania mogą zmienić sytuację tak, że zadanie nie będzie mogło wykonywać albo nie będzie zainteresowane wykonaniem danej operacji w tym właśnie momencie. Wtedy zadanie również powinno ponownie zawiesić swoje wykonanie wywołaniem `wait()`.
- Może też się zdarzyć, że na blokadzie danego obiektu oczekiwać będzie kilka zadań, zawieszonych z *różnych* powodów (w którym to przypadku *trzeba* je wybudzać wywołaniem `notifyAll()`). W takim przypadku w zadaniu należy sprawdzić, czy wybudzenie miało właściwe podstawy i ewentualnie ponownie wywołać `wait()`.

Jak widać, kontrola zajścia oczekiwanych okoliczności i ewentualne ponowne zawieszenie w `wait()` ma zasadnicze znaczenie. Zwykle model ten wciela się w życie właśnie w postaci pętli `while`, gdzie oczekiwane okoliczności są wyrażone w warunku kontynuacji pętli.

Ćwiczenie 21. Utwórz dwie implementacje `Runnable`: pierwsza w metodzie `run()` ma inicjować działanie i wywoływać `wait()`. Druga powinna przechwytywać referencję obiektu pierwszej klasy. Jej metoda `run()` powinna po upływie pewnej ilości czasu wywołać `notifyAll()` dla pierwszego zadania tak, aby mogło ono wypisać komunikat na konsoli. Przetestuj klasy z użyciem wykonawcy (2).

Ćwiczenie 22. Stwórz przykład „aktywnego oczekiwania". Jeden z wątków zasypia na chwilę, po czym ustawia flagę na `true`. Drugi z wątków obserwuje flagę w pętli `while` (to jest właśnie „aktywne oczekiwanie"), a gdy jej wartość zmieni się na `true`, zmienia ją z powrotem na `false` i wyświetla komunikat o zmianie. Zwróć uwagę, ile czasu program niepotrzebnie spędza na aktywnym oczekiwaniu, i stwórz drugą wersję programu wykorzystującą `wait()` zamiast aktywnego oczekiwania (4).

Utracone sygnały

Kiedy koordynujesz dwa wątki przy użyciu metod `wait()`-`notify()` albo `wait()`-`notifyAll()`, ryzykujesz przegapienie sygnału. Załóżmy, że T1 to wątek kierujący powiadomienie do wątku T2 i że oba wątki są zaimplementowane na bazie poniższego (nie najlepszego) schematu:

```
T1:
synchronized(sharedMonitor) {
  <ustawienie okoliczności dla T2>
  sharedMonitor.notify();
}

T2:
while(oczekiwane-okoliczności) {
  // Punkt 1
  synchronized(sharedMonitor) {
    sharedMonitor.wait();
  }
}
```

Ustawianie okoliczności dla T2 to czynność mająca odwieść T2 od wywołania `wait()`, jeśli jeszcze do niego nie doszło.

Załóżmy, że T2 sprawdza oczekiwane okoliczności i stwierdza, że już zaszły. W punkcie 1. planista może jednak przełączyć kontekst do zadania T1. T1 przeprowadzi operację zmiany okoliczności dla T2 i wywoła `notify()`. Kiedy zadanie T2 wznowi wykonanie, będzie już za późno na zorientowanie się co do zmiany okoliczności i zadanie wywoła `wait()`. Tym samym powiadomienie `notify()` zostanie przegapione, a zadanie T2 będzie w nieskończoność oczekiwało na sygnał, który został wysłany już wcześniej. Dojdzie więc do klasycznego zakleszczenia zadań.

Rozwiązanie polega na zapobieżeniu sytuacji hazardowej przy oczekiwanych okolicznościach. Oto właściwa implementacja T2:

```
synchronized(sharedMonitor) {
  while (oczekiwane-okoliczności)
    sharedMonitor.wait();
}
```

Teraz, jeśli T1 wykona się jako pierwsze, to gdy sterowanie powróci do zadania T2, zadanie to wykryje zmianę okoliczności i *nie wywoła* `wait()`. A jeśli jako pierwsze uruchomi się zadanie T2, to rozpocznie ono oczekiwanie w `wait()` i później zostanie wybudzone przez T1. Nie ma wtedy możliwości utracenia sygnału.

notify() kontra notifyAll()

Ponieważ technicznie nic nie uniemożliwia sytuacji, w której na wspólnym obiekcie `Car` oczekuje (w zawieszeniu wymuszonym wywołaniem `wait()`) wiele zadań, niemal zawsze lepiej wywoływać `notifyAll()` w miejsce `notify()`. Ale struktura ostatnio omawianego programu jest akurat taka, że oczekiwanie będzie dotyczyć zawsze tylko jednego zadania, więc można tu bezpiecznie użyć `notify()` zamiast `notifyAll()`.

Stosowanie `notify()` w miejsce `notifyAll()` to rodzaj optymalizacji. Skoro spośród zadań zawieszonych na danej blokadzie wywołanie `notify()` wybudzi tylko jedno, to trzeba zadbać o to, aby było to zadanie właściwe. Skuteczne użycie `notify()` wymaga więc, aby wszystkie oczekujące na danej blokadzie zadania oczekiwały na te same okoliczności, ponieważ zróżnicowanie przyczyn zawieszenia uniemożliwi wybranie właściwego zadania do wybudzenia. Przy wywołaniu `notify()` na zajściu oczekiwanych okoliczności skorzysta tylko jedno z oczekujących zadań. Warunek ujednolicenia oczekiwanych okoliczności dotyczy również wszelkich podklas klasy zadań. Jeśli nie można tego zagwarantować, należy korzystać z `notifyAll()`.

W dyskusjach na temat wielowątkowości w Javie pada często jakże mylące twierdzenie, że `notifyAll()` wybudza „wszystkie oczekujące zadania". Czy ma to oznaczać, że wybudzanie `notifyAll()` dotyczy wszystkich zadań zawieszonych wywołaniem `wait()` niezależnie od miejsca w programie? Poniższy kod pokazuje, że to nieprawda — w rzeczywistości wybudzane są tylko te zadania, które zostały *zawieszone na blokadzie*, na rzecz której nastąpiło wywołanie `notifyAll()`:

```
//: concurrency/NotifyVsNotifyAll.java
import java.util.concurrent.*;
import java.util.*;

class Blocker {
```

```
  synchronized void waitingCall() {
    try {
      while(!Thread.interrupted()) {
        wait();
        System.out.print(Thread.currentThread() + " ");
      }
    } catch(InterruptedException e) {
      // Można tak opuścić metodę
    }
  }
  synchronized void prod() { notify(); }
  synchronized void prodAll() { notifyAll(); }
}

class Task implements Runnable {
  static Blocker blocker = new Blocker();
  public void run() { blocker.waitingCall(); }
}

class Task2 implements Runnable {
  // Osobny egzemplarz klasy Blocker:
  static Blocker blocker = new Blocker();
  public void run() { blocker.waitingCall(); }
}

public class NotifyVsNotifyAll {
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new Task());
    exec.execute(new Task2());
    Timer timer = new Timer();
    timer.scheduleAtFixedRate(new TimerTask() {
      boolean prod = true;
      public void run() {
        if(prod) {
          System.out.print("\nnotify() ");
          Task.blocker.prod();
          prod = false;
        } else {
          System.out.print("\nnotifyAll() ");
          Task.blocker.prodAll();
          prod = true;
        }
      }
    }, 400, 400); // Co 0,4 sekundy
    TimeUnit.SECONDS.sleep(5); // Niech się kręci...
    timer.cancel();
    System.out.println("\nOdwołanie zegara");
    TimeUnit.MILLISECONDS.sleep(500);
    System.out.print("Task2.blocker.prodAll() ");
    Task2.blocker.prodAll();
    TimeUnit.MILLISECONDS.sleep(500);
    System.out.println("\nZamykanie");
    exec.shutdownNow(); // Przerwanie wszystkich zadań
  }
} /* Output: (Sample)
notify() Thread[pool-1-thread-1,5,main]
```

```
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-5,5,main] Thread[pool-1-thread-
4,5,main] Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-2,5,main] Thread[pool-1-thread-
3,5,main] Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-5,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-5,5,main] Thread[pool-1-thread-
4,5,main] Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-2,5,main] Thread[pool-1-thread-
3,5,main] Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-5,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-5,5,main] Thread[pool-1-thread-
4,5,main] Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-2,5,main]
notify() Thread[pool-1-thread-1,5,main]
notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-2,5,main] Thread[pool-1-thread-
3,5,main] Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-5,5,main]
Odwołanie zegara
Task2.blocker.prodAll() Thread[pool-1-thread-6,5,main]
Zamykanie
*///:~
```

Zadania `Task` i `Task2` dysponują własnymi egzemplarzami klasy `Blocker`, więc każde z zadań klasy `Task` podlega zawieszaniu na blokadzie `Task.blocker`, a zawieszenia zadań `Task2` są regulowane blokadą `Task2.blocker`. W metodzie `main()` tworzony jest obiekt klasy `java.util.Timer` konfigurowany tak, aby co 4 dziesiąte sekundy uruchamiać metodę `run()`, a ta z kolei przełącza się pomiędzy wywołaniami `notify()` i `notifyAll()` dla blokady `Task.blocker`, wybierając odpowiednie metody `prod...` klasy `Task`.

Na wyjściu programu widać, że choć obiekt `Task2` istnieje i został zawieszony na blokadzie `Task2.blocker`, to wywołania `notify()` i `notifyAll()` na rzecz blokady `Task.blocker` w ogóle go nie dotyczą — ani razu nie prowadzą do wybudzenia zadania `Task2`. Podobnie, pod koniec metody `main()` następuje wywołanie `cancel()` na rzecz obiektu `timer`, ale mimo odwołania mechanizmu zegarowego pięć zadań wciąż działa, trwając w wywołaniach `Task.blocker.waitingCall()`. Z kolei wyjście spowodowane wywołaniem `Task2.blocker.prodAll()` nie dotyczy żadnego z zadań zawieszonych na blokadzie `Task.blocker`.

Ma to jak najbardziej sens — widać go również w metodach `prod()` i `prodAll()` klasy `Blocker`. Metody te są synchronizowane, co oznacza, że zakładają swoją własną blokadę, więc kiedy wywołują `notify()` czy `notifyAll()`, logicznym jest, że wywołania te dotyczą właśnie tej blokady — i wybudzają jedynie zadania zawieszone na tej konkretnej blokadzie.

Metoda `Blocker.waitingCall()` jest na tyle prosta, że pętlę `while(!Thread.interrupted())` można by zastąpić nieskończoną pętlą `for(;;)` i uzyskać ten sam efekt, bo w prezentowanym przykładzie nie ma różnicy pomiędzy opuszczeniem pętli wymuszonym wyjątkiem a opuszczeniem dobrowolnym, w wyniku sprawdzenia stanu znacznika przerwania zadania (w obu przypadkach przerwanie jest tak samo obsługiwane). Ale dla porządku przykład sprawdza metodą `interrupted()` ewentualne przerwanie. Jeśli kiedyś ktoś zdecyduje się rozbudować pętlę, brak pokrycia obu ścieżek przerwania byłby już ryzykowny.

Ćwiczenie 23. Pokaż, że program *WaxOMatic.java* zadziała równie dobrze, jeśli zamiast `notifyAll()` użyjesz `notify()` (7).

Producenci i konsumenci

Wyobraźmy sobie restaurację, w której pracuje jeden kucharz i jeden kelner. Kelner musi czekać, aż kucharz przygotuje posiłek. Gdy posiłek jest gotowy, kucharz informuje o tym kelnera, który odbiera posiłek, przekazuje na salę i znowu czeka. Jest to doskonały przykład współdziałania zadań: kucharz reprezentuje *producenta*, a kelner — *konsumenta*. Oba zadania muszą wymieniać komunikaty o dostępności i odebraniu posiłku, a cały system powinien dawać się elegancko zamykać. Poniżej przedstawiłem ten przykład wyrażony w formie kodu:

```
// : concurrency/Restaurant.java
// Współdziałanie zadań według modelu producent-konsument.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Meal {
  private final int orderNum;
  public Meal(int orderNum) { this.orderNum = orderNum; }
  public String toString() { return "Danie " + orderNum; }
}

class WaitPerson implements Runnable {
  private Restaurant restaurant;
  public WaitPerson(Restaurant r) { restaurant = r; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        synchronized(this) {
          while(restaurant.meal == null)
            wait(); // ... oczekiwanie na efekty pracy kucharza
        }
        print("Kelner odebrał danie " + restaurant.meal);
        synchronized(restaurant.chef) {
          restaurant.meal = null;
          restaurant.chef.notifyAll(); // Gotowy do następnej tury
        }
      }
    } catch(InterruptedException e) {
      print("Przerwano zadanie kelnera");
    }
  }
}

class Chef implements Runnable {
  private Restaurant restaurant;
  private int count = 0;
  public Chef(Restaurant r) { restaurant = r; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        synchronized(this) {
          while(restaurant.meal != null)
            wait(); // ... oczekiwanie na odbiór dania
        }
        if(++count == 10) {
          print("Koniec zapasów, zamykamy");
          restaurant.exec.shutdownNow();
```

```
        }
        printnb("Zrobione! ");
        synchronized(restaurant.waitPerson) {
          restaurant.meal = new Meal(count);
          restaurant.waitPerson.notifyAll();
        }
        TimeUnit.MILLISECONDS.sleep(100);
      }
    } catch(InterruptedException e) {
      print("Przerwano zadanie kucharza");
    }
  }
}

public class Restaurant {
  Meal meal;
  ExecutorService exec = Executors.newCachedThreadPool();
  WaitPerson waitPerson = new WaitPerson(this);
  Chef chef = new Chef(this);
  public Restaurant() {
    exec.execute(chef);
    exec.execute(waitPerson);
  }
  public static void main(String[] args) {
    new Restaurant();
  }
} /* Output:
Zrobione! Kelner odebrał danie 1
Zrobione! Kelner odebrał danie 2
Zrobione! Kelner odebrał danie 3
Zrobione! Kelner odebrał danie 4
Zrobione! Kelner odebrał danie 5
Zrobione! Kelner odebrał danie 6
Zrobione! Kelner odebrał danie 7
Zrobione! Kelner odebrał danie 8
Zrobione! Kelner odebrał danie 9
Koniec zapasów, zamykamy
Przerwano zadanie kelnera
Zrobione! Przerwano zadanie kucharza
*///:~
```

Tak kucharz (`Chef`), jak i kelner (`WaitPerson`) muszą wiedzieć, w jakiej restauracji (`Restaurant`) pracuje, ponieważ muszą dostarczać i odbierać dania do i z „okienka kuchennego" — `restaurant.meal`. W metodzie `run()` `WaitPerson` przechodzi w stan `wait()`, zatrzymując zadanie aż do momentu obudzenia go wywołaniem `notify()`. Ponieważ jest to bardzo prosty program, wiemy, że tylko jedno zadanie będzie czekać na blokadzie obiektu `WaitPerson` — samo zadanie `WaitPerson`. Dlatego wywołanie metody `notify()` jest bezpieczne. W bardziej złożonych sytuacjach na blokadzie konkretnego obiektu może czekać kilka zadań, a zatem nie wiadomo, który z nich należy obudzić. Rozwiązaniem tego problemu jest wywołanie metody `notifyAll()`, która budzi wszystkie zadania oczekujące na danej blokadzie. Każdy z obudzonych wątków musi następnie zdecydować, czy przekazana informacja ma dla niego znaczenie.

Kiedy kucharz dostarczy danie i powiadomi o tym kelnera, sam zaczyna oczekiwanie, aż kelner odbierze posiłek i dostarczy go na salę, po czym powiadomi o tym kucharza, aby ten mógł przystąpić do przygotowywania następnego dania.

Zauważ, że wywołanie `wait()` zostało umieszczone w pętli `while`, sprawdzającej ten sam warunek, na który czekamy. Z początku wydaje się to dziwne — przecież jeśli czekamy na danie, to po obudzeniu zadania musi ono być dostępne, prawda? Jednak problem polega na tym, że w aplikacjach współbieżnych podczas budzenia zadania `WaitPerson` jakieś inne zadanie może się wtrącić i przechwycić danie. Jedynym bezpiecznym rozwiązaniem jest *konsekwentne* wywoływanie `wait()` w następujący sposób (z odpowiednią synchronizacją i zabezpieczeniem przed utratą sygnałów):

```
while(warunekNieJestSpełniony)
  wait();
```

Rozwiązanie to gwarantuje, że przed opuszczeniem pętli oczekiwania warunek zostanie spełniony; jeśli natomiast zadanie zostanie powiadomione o czymś, co nie dotyczy warunku (co się może zdarzyć w przypadku wywołania `notifyAll()`), lub gdy warunek zmieni się przed całkowitym obudzeniem zadania, mamy gwarancję, że zadanie ponownie przejdzie w stan oczekiwania.

Zauważ, że wywołania `notifyAll()` muszą w pierwszej kolejności pozyskać blokadę `WaitPerson`. Jest to możliwe, gdyż wywołanie `wait()` w `WaitPerson.run()` automatycznie zwalnia blokadę. Ponieważ w celu wywołania `notifyAll()` konieczne jest posiadanie blokady, mamy gwarancję, że dwa wątki starające się wywołać `notifyAll()` nie będą sobie wzajemnie przeszkadzać.

Metody `run()` obu zadań są zaprojektowane z uwzględnieniem właściwej obsługi przerwania zadań, co polega na ujęciu całości ciała metody `run()` w bloku kodu chronionego `try`. Klauzula `catch` uzupełniająca taki blok kończy się tuż przed nawiasem klamrowym zamykającym ciało metody — jeśli więc zadanie otrzyma wyjątek `InterruptedException`, zakończy działanie natychmiast po jego przechwyceniu.

W zadaniu `Chef` po wywołaniu `shutdownNow()` *można* po prostu powrócić z metody `run()` i normalnie właśnie tak powinno się kończyć zadanie. Tyle że jest tu jeszcze mała ciekawostka. Otóż wywołanie `shutdownNow()` wymusza wysłanie komunikatu `interrupt()` do wszystkich wątków zarządzanych przez danego wykonawcę. Ale w przypadku zadania `Chef` nie dochodzi do natychmiastowego przerwania zadania w wyniku odebrania komunikatu `interrupt()`, bo komunikat ten powoduje zgłoszenie wyjątku tylko wtedy, kiedy zadanie wykonuje dającą się przerywać operację blokującą. Dlatego też na wyjściu pojawia się jeszcze komunikat „Zrobione!", a wyjątek `InterruptedException` pojawia się dopiero wtedy, kiedy zadanie próbuje się uśpić wywołaniem `sleep()`. Gdybyśmy usunęli wywołanie `sleep()` z pętli, zadanie dotarłoby na szczyt pętli metody `run()` i zakończyło działanie w wyniku samodzielnej kontroli znacznika przerwania `Thread.interrupted()`, bez wyjątku.

W powyższym programie jest tylko jedno miejsce, w którym jeden wątek może zapisać obiekt, tak że drugi wątek może go później odebrać. W typowych przykładach producent-konsument do przechowywania produkowanych i konsumowanych obiektów wykorzystywana jest kolejka typu „pierwszy wstawiony, pierwszy odebrany" (FIFO). Więcej na ten temat dowiesz się w dalszej części rozdziału.

Ćwiczenie 24. Rozwiąż modelowy problem jednego konsumenta i jednego producenta za pomocą metod `wait()` i `notifyAll()`. Producent nie może przepełnić bufora konsumenta, a więc nie może produkować szybciej, niż konsument konsumuje; jeśli z kolei

konsument jest szybszy w konsumowaniu, powinieneś zadbać o to, aby w żadnym razie nie pobrał tych samych danych więcej niż raz (co byłoby odpowiednikiem niedopełnienia bufora producenta). Nie powinieneś przyjmować żadnych założeń co do względnej szybkości wykonania zadań producenta i konsumenta (1).

Ćwiczenie 25. W klasie `Chef` z programu *Restaurant.java* zastosuj powrót (instrukcją `return`) z metody `run()` po wywołaniu `shutdownNow()` i zaobserwuj różnicę w zachowaniu programu (1).

Ćwiczenie 26. Dodaj do programu *Restaurant.java* klasę zadania pomywacza — `BusBoy`. Pomywacz powinien oczekiwać na powiadomienie od kelnera, a następnie przystępować do sprzątania (8).

Jawne blokady i obiekty Condition

W bibliotece `java.util.concurrent` w wydaniu SE5 mamy do dyspozycji jawne narzędzia, które możemy wykorzystać do przepisania programu *WaxOMatic.java*. Rolę podstawowej klasy wykorzystującej muteks i pozwalającej zadaniu na wejście w stan zawieszenia pełni klasa `Condition`. Zawieszenie wymusza się wywołaniem metody `await()` na rzecz obiektu `Condition`. Kiedy zewnętrzne okoliczności umożliwią ponowne podjęcie wykonania zadania, należy je o tym powiadomić wywołaniem metody `signal()` (jeśli chcemy wybudzić jedno zadanie) albo `signalAll()` (jeśli chcemy wybudzić wszystkie zadania zawieszone na danym egzemplarzu `Condition`); podobnie jak w przypadku `notify()` i `notifyAll()`, w ogólnym przypadku bezpieczniejsze jest wywołanie `signalAll()`.

Oto program *WaxOMatic.java* z metodami `waitForWaxing()` i `waitForBuffing()` przepisanymi pod kątem klasy `Condition`:

```
// : concurrency/waxomatic2/WaxOMatic2.java
// Zastosowanie obiektów Lock i Condition.
package concurrency.waxomatic2;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class Car {
  private Lock lock = new ReentrantLock();
  private Condition condition = lock.newCondition();
  private boolean waxOn = false;
  public void waxed() {
    lock.lock();
    try {
      waxOn = true; // Gotowy do polerowania
      condition.signalAll();
    } finally {
      lock.unlock();
    }
  }
  public void buffed() {
    lock.lock();
    try {
      waxOn = false; // Gotowy do nałożenia następnej warstwy wosku
      condition.signalAll();
    } finally {
```

```
      lock.unlock();
    }
  }
  public void waitForWaxing() throws InterruptedException {
    lock.lock();
    try {
      while(waxOn == false)
        condition.await();
    } finally {
      lock.unlock();
    }
  }
  public void waitForBuffing() throws InterruptedException{
    lock.lock();
    try {
      while(waxOn == true)
        condition.await();
    } finally {
      lock.unlock();
    }
  }
}

class WaxOn implements Runnable {
  private Car car;
  public WaxOn(Car c) { car = c; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        printnb("Woskowanie! ");
        TimeUnit.MILLISECONDS.sleep(200);
        car.waxed();
        car.waitForBuffing();
      }
    } catch(InterruptedException e) {
      print("Wyjście wymuszone przerwaniem");
    }
    print("Koniec zadania woskowania");
  }
}

class WaxOff implements Runnable {
  private Car car;
  public WaxOff(Car c) { car = c; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        car.waitForWaxing();
        printnb("Polerowanie! ");
        TimeUnit.MILLISECONDS.sleep(200);
        car.buffed();
      }
    } catch(InterruptedException e) {
      print("Wyjście wymuszone przerwaniem");
    }
    print("Koniec zadania polerowania");
  }
}
```

```
public class WaxOMatic2 {
  public static void main(String[] args) throws Exception {
    Car car = new Car();
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(new WaxOff(car));
    exec.execute(new WaxOn(car));
    TimeUnit.SECONDS.sleep(5);
    exec.shutdownNow();
  }
} /* Output: (90% match)
Woskowanie! Polerowanie! Woskowanie! Polerowanie! Woskowanie! Polerowanie! Woskowanie!
Polerowanie! Woskowanie! Polerowanie! Woskowanie! Polerowanie! Woskowanie! Polerowanie!
Woskowanie! Polerowanie! Woskowanie! Polerowanie! Woskowanie! Polerowanie! Woskowanie!
Polerowanie! Woskowanie! Polerowanie! Woskowanie! Wyjście wymuszone przerwaniem
Koniec zadania polerowania
Wyjście wymuszone przerwaniem
Koniec zadania woskowania
*///:~
```

W konstruktorze egzemplarza `Car` pojedynczy obiekt jawnej blokady `Lock` generuje obiekt `Condition`, który służy potem do zarządzania komunikacją między zadaniami. Jednakże obiekt `Condition` nie zawiera żadnych informacji o stanie procesu, więc trzeba go sygnalizować dodatkowymi elementami danych, których rolę pełni tu pole `waxOn` (typu `boolean`).

Każde wywołanie metody `lock()` musi być bezpośrednio uzupełnione blokiem `try-finally`, który zagwarantuje zdjęcie blokady niezależnie od okoliczności. Podobnie jak w przypadku wersji z blokadami wbudowanymi, zadanie, które chce wywołać metody `await()`, `signal()` czy `signalAll()`, musi uprzednio pozyskać (założyć) blokadę.

Zauważ, że to rozwiązanie jest bardziej złożone od poprzedniego, a wzrost poziomu komplikacji nie daje tu zupełnie żadnych korzyści. Obiekty `Lock` i `Condition` przydają się dopiero w rozwiązaniach trudniejszych problemów wielowątkowości.

Ćwiczenie 27. Zmodyfikuj program *Restaurant.java* tak, aby korzystał z jawnych blokad `Lock` i obiektów `Condition` (2).

Producenci, konsumenci i kolejki

Metody `wait()` i `notifyAll()` rozwiązują problem synchronizacji współpracujących zadań, ale jest to rozwiązanie stosunkowo niskopoziomowe, wymagające wymiany bezpośredniej komunikatów na każdym etapie współpracy. Tymczasem w wielu przypadkach można wspiąć się na wyższy poziom abstrakcji i rozwiązywać problemy kooperacji zadań za pośrednictwem *synchronizowanych kolejek*, które umożliwiają dostęp do elementów tylko jednemu zadaniu naraz. Kolejki takie są udostępniane za pośrednictwem interfejsu `java.util.concurrent.BlockingQueue`, posiadającego wiele standardowych implementacji. Jedną z popularniejszych jest `LinkedBlockingQueue`, stanowiąca kolejkę o nieograniczonej liczbie elementów; z kolei `ArrayBlockingQueue` to kolejka o ustalonej z góry i niezmiennej liczbie elementów — próba umieszczania nadmiarowych elementów powoduje zawieszenie zadania wstawiającego.

Omawiane kolejki zawieszają również zadanie konsumenta, który chciałby wydobyć obiekt z kolejki, gdy ta jest pusta — zawieszone zadanie jest wznawiane dopiero wtedy, kiedy w kolejce pojawią się nowe elementy. Kolejki synchronizowane mogą posłużyć do rozwiązania znacznej liczby problemów kooperacji w prostszy i bardziej niezawodny sposób niż przy samodzielnym stosowaniu metod `wait()` i `notifyAll()`.

Oto prosty test, w ramach którego uszeregujemy wykonanie egzemplarzy zadań `LiftOff`. Rolę konsumenta będzie pełnił obiekt klasy `LiftOffRunner`, który będzie wyciągał z kolejki kolejne obiekty `LiftOff` i bezpośrednio inicjował ich wykonanie (to znaczy będzie wykonywał metodę `run()` kolejnych zadań we własnym wątku).

```
//: concurrency/TestBlockingQueues.java
// {RunByHand}
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class LiftOffRunner implements Runnable {
  private BlockingQueue<LiftOff> rockets;
  public LiftOffRunner(BlockingQueue<LiftOff> queue) {
    rockets = queue;
  }
  public void add(LiftOff lo) {
    try {
      rockets.put(lo);
    } catch(InterruptedException e) {
      print("Przerwanie w metodzie put()");
    }
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        LiftOff rocket = rockets.take();
        rocket.run(); // Uruchomienie zadania we własnym wątku
      }
    } catch(InterruptedException e) {
      print("Obudzony z take()");
    }
    print("Kończenie zadania LiftOffRunner");
  }
}

public class TestBlockingQueues {
  static void getkey() {
    try {
      // Kompensacja różnicy pomiędzy systemami Windows i Linux
      // odnośnie znaku generowanego przez naciśnięcie klawisza Enter:
      new BufferedReader(
        new InputStreamReader(System.in)).readLine();
    } catch(java.io.IOException e) {
      throw new RuntimeException(e);
    }
  }
  static void getkey(String message) {
    print(message);
    getkey();
```

```
  }
  static void
  test(String msg, BlockingQueue<LiftOff> queue) {
    print(msg);
    LiftOffRunner runner = new LiftOffRunner(queue);
    Thread t = new Thread(runner);
    t.start();
    for(int i = 0; i < 5; i++)
      runner.add(new LiftOff(5));
    getkey("Naciśnij 'Enter' (" + msg + ")");
    t.interrupt();
    print("Koniec testu " + msg);
  }
  public static void main(String[] args) {
    test("LinkedBlockingQueue", // Nieograniczony rozmiar
      new LinkedBlockingQueue<LiftOff>());
    test("ArrayBlockingQueue", // Stały rozmiar
      new ArrayBlockingQueue<LiftOff>(3));
    test("SynchronousQueue", // Kolejka jednoelementowa
      new SynchronousQueue<LiftOff>());
  }
} ///:~
```

Zadania odliczania (`LiftOff`) są w metodzie `main()` umieszczane w kolejnych rodzajach kolejek synchronizowanych, z których odbiera je działające współbieżnie zadanie `LiftOffRunner`. Zauważ, że `LiftOffRunner` może zignorować kwestie synchronizacji z wątkiem metody `main()`, bo zajmuje się nimi obiekt kolejki.

Ćwiczenie 28. Zmień program *TestBlockingQueue.java*, dodając nowe zadanie, które przejmie od `main()` odpowiedzialność za zasilanie kolejki synchronizowanej zadaniami `LiftOff` (3).

Kolejka do tostów

W ramach kolejnego przykładu zastosowań kolejek synchronizowanych z hierarchii `BlockingQueue` zasymulujemy maszynę, która realizuje trzy zadania: opieka tosty, smaruje je masłem, a potem smaruje dżemem. Tosty, jak na taśmie produkcyjnej, będziemy przekazywać do kolejnych etapów za pośrednictwem kolejek synchronizowanych:

```
//: concurrency/ToastOMatic.java
// Toster - taśma produkcyjna.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Toast {
  public enum Status { DRY, BUTTERED, JAMMED }
  private Status status = Status.DRY;
  private final int id;
  public Toast(int idn) { id = idn; }
  public void butter() { status = Status.BUTTERED; }
  public void jam() { status = Status.JAMMED; }
  public Status getStatus() { return status; }
  public int getId() { return id; }
  public String toString() {
    return "Tost " + id + ": " + status;
```

```
  }
}

class ToastQueue extends LinkedBlockingQueue<Toast> {}

class Toaster implements Runnable {
  private ToastQueue toastQueue;
  private int count = 0;
  private Random rand = new Random(47);
  public Toaster(ToastQueue tq) { toastQueue = tq; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        TimeUnit.MILLISECONDS.sleep(
          100 + rand.nextInt(500));
        // Opiekanie tosta
        Toast t = new Toast(count++);
        print(t);
        // Wstawienie do kolejki
        toastQueue.put(t);
      }
    } catch(InterruptedException e) {
      print("Przerwano zadanie tostowania");
    }
    print("Toster wyłączony");
  }
}

// Smarowanie tosta masłem:
class Butterer implements Runnable {
  private ToastQueue dryQueue, butteredQueue;
  public Butterer(ToastQueue dry, ToastQueue buttered) {
    dryQueue = dry;
    butteredQueue = buttered;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // Blokowanie do czasu opieczenia następnego tosta:
        Toast t = dryQueue.take();
        t.butter();
        print(t);
        butteredQueue.put(t);
      }
    } catch(InterruptedException e) {
      print("Przerwano smarowanie masłem");
    }
    print("Smarownica masła wyłączona");
  }
}

// Smarowanie dżemem:
class Jammer implements Runnable {
  private ToastQueue butteredQueue, finishedQueue;
  public Jammer(ToastQueue buttered, ToastQueue finished) {
    butteredQueue = buttered;
    finishedQueue = finished;
  }
```

```
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // Blokowanie do czasu dostępności następnego tosta:
        Toast t = butteredQueue.take();
        t.jam();
        print(t);
        finishedQueue.put(t);
      }
    } catch(InterruptedException e) {
      print("Przerwano smarowanie dżemem");
    }
    print("Smarownica dżemu wyłączona");
  }
}

// Konsumpcja tosta:
class Eater implements Runnable {
  private ToastQueue finishedQueue;
  private int counter = 0;
  public Eater(ToastQueue finished) {
    finishedQueue = finished;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // Blokowanie w oczekiwaniu na przygotowanie tosta:
        Toast t = finishedQueue.take();
        // Sprawdzenie, czy tost przybył w odpowiedniej kolejności,
        // i czy jest posmarowany masłem i dżemem:
        if(t.getId() != counter++ ||
           t.getStatus() != Toast.Status.JAMMED) {
          print(">>>> Błąd: " + t);
          System.exit(1);
        } else
          print("Chrup! " + t);
      }
    } catch(InterruptedException e) {
      print("Przerwano śniadanie");
    }
    print("Biesiadnik wyłączony");
  }
}

public class ToastOMatic {
  public static void main(String[] args) throws Exception {
    ToastQueue dryQueue = new ToastQueue(),
               butteredQueue = new ToastQueue(),
               finishedQueue = new ToastQueue();
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(new Toaster(dryQueue));
    exec.execute(new Butterer(dryQueue, butteredQueue));
    exec.execute(new Jammer(butteredQueue, finishedQueue));
    exec.execute(new Eater(finishedQueue));
    TimeUnit.SECONDS.sleep(5);
    exec.shutdownNow();
  }
} /* (Execute to see output) *///:~
```

Klasa Toast znakomicie ilustruje przydatność typów wyliczeniowych. Zauważ, że nie ma tu jawnej synchronizacji zadań (ani za pomocą obiektów blokad, ani za pomocą metod synchronizowanych), bo synchronizacją zajmują się niejawnie kolejki (synchronizowane wewnętrznie); brak synchronizacji wynika też z projektu systemu — każdy tost jest w danym momencie obrabiany przez tylko jedno zadanie. Z racji synchronizacji kolejek zadania są zawieszane i wznawiane samoczynnie. Mamy tu do czynienia z dramatycznym uproszczeniem symulacji. Powiązanie pomiędzy klasami, które istniało wcześniej w postaci wywołań `wait()` i `notifyAll()`, zostało całkowicie wyeliminowane, bo klasy zadań komunikują się wyłącznie za pośrednictwem kolejek.

Ćwiczenie 29. Zmodyfikuj program *ToastOMatic.java* tak, aby toster przygotowywał tosty składane z masłem orzechowym i tosty składane z dżemem. Alternatywne odmiany tostów powinny być przygotowywane na osobnych liniach, które łączą się na wyjściu tostera (8).

Przekazywanie danych pomiędzy zadaniami za pomocą potoków

Często przydatne jest przekazywanie informacji pomiędzy wątkami przy wykorzystaniu operacji wejścia-wyjścia. Biblioteki mogą udostępniać wsparcie dla operacji wejścia-wyjścia pomiędzy wątkami w formie *potoków* (ang. *pipes*). W bibliotece wejścia-wyjścia Javy potoki zostały zaimplementowane w klasie `PipedWriter` (umożliwiającej wątkowi zapis danych do potoku) oraz `PipedReader` (umożliwiającej innemu wątkowi odczyt danych z tego samego potoku). Można to sobie wyobrazić jako pewną odmianę modelu producent-konsument, w którym potok jest standardowym rozwiązaniem. Potok to odmiana kolejki synchronizowanej dostępna w wydaniach Javy przed wprowadzeniem klas `BlockingQueue`.

Oto prosty przykład, w którym dwa wątki przekazują sobie informacje przy wykorzystaniu potoków:

```
//: concurrency/PipedIO.java
// Potoki w służbie wymiany danych pomiędzy zadaniami
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Sender implements Runnable {
  private Random rand = new Random(47);
  private PipedWriter out = new PipedWriter();
  public PipedWriter getPipedWriter() { return out; }
  public void run() {
    try {
      while(true)
        for(char c = 'A'; c <= 'z'; c++) {
          out.write(c);
          TimeUnit.MILLISECONDS.sleep(rand.nextInt(500));
        }
    } catch(IOException e) {
      print(e + " wyjątek zapisu w klasie Sender");
    } catch(InterruptedException e) {
      print(e + " przerwano zadanie Sender");
```

```
    }
  }
}

class Receiver implements Runnable {
  private PipedReader in;
  public Receiver(Sender sender) throws IOException {
    in = new PipedReader(sender.getPipedWriter());
  }
  public void run() {
    try {
      while(true) {
        // Blokowanie do czasu pojawienia się nowych znaków:
        printnb("Odczyt: " + (char)in.read() + ", ");
      }
    } catch(IOException e) {
      print(e + " wyjątek odczytu w klasie Receiver");
    }
  }
}

public class PipedIO {
  public static void main(String[] args) throws Exception {
    Sender sender = new Sender();
    Receiver receiver = new Receiver(sender);
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(sender);
    exec.execute(receiver);
    TimeUnit.SECONDS.sleep(4);
    exec.shutdownNow();
  }
} /* Output: (65% match)
Odczyt: A, Odczyt: B, Odczyt: C, Odczyt: D, Odczyt: E, Odczyt: F, Odczyt: G, Odczyt: H, Odczyt: I,
Odczyt: J, Odczyt: K, Odczyt: L, Odczyt: M, java.lang.InterruptedException: sleep interrupted przerwano
zadanie Sender
java.io.InterruptedIOException wyjątek odczytu w klasie Receiver
*///:~
```

Klasy `Sender` oraz `Receiver` reprezentują zadania, które muszą się wzajemnie komunikować. `Sender` tworzy samodzielny obiekt `PipedWriter`, jednak w `Receiver`, tworząc obiekt `PipedReader`, trzeba go skojarzyć z `PipedWriter` (przekazując go w konstruktorze `PipedReader`). `Sender` zapisuje dane w potoku, posługując się obiektem `Writer`, po czym zasypia na losowy czas. Jednak w obiekcie `Receiver` nie ma żadnych wywołań metody `sleep()` ani `wait()`. Jednak wywołanie `read()` automatycznie blokuje zadanie, gdy w potoku nie ma żadnych danych do odczytu.

Zauważ, że zadania `sender` i `receiver` są uruchamiane w metodzie `main()` *po* całkowitym utworzeniu obiektów. Uruchomienie ich w przypadku, gdy tworzenie obiektów nie zostało w pełni zakończone, może spowodować niespójne działanie potoków na niektórych platformach systemowych (kolejki synchronizowane są więc solidniejsze i prostsze w użyciu).

Przy okazji wywołania `shutdownNow()` ujawnia się ważna różnica pomiędzy operacjami `PipedReader` a normalnymi operacjami wejścia-wyjścia — otóż operacja odczytu z `PipedReader` jest operacją przerywalną; gdybyśmy zamienili wywołanie `in.read()` na `System.in.read()`, przerwanie `interrupt()` nie dałoby rady wybudzić wątku zawieszonego w metodzie `read()`.

Ćwiczenie 30. Zmodyfikuj program *PipedIO.java* tak, aby zamiast potoków wykorzystywał kolejki synchronizowane typu `BlockingQueue` (1).

Zakleszczenie

Wiesz już, że obiekt może posiadać metody synchronizowane i inne mechanizmy blokowania uniemożliwiające dostęp do obiektu bez założenia blokady. Wiesz też, że zadania mogą być blokowane. Istnieje więc możliwość, że jedno zadanie utknie, oczekując na inne zadanie, które czeka na kolejne zadanie itd., aż łańcuszek powróci do zadania czekającego na to pierwsze. Otrzymamy zamkniętą pętlę zadań czekających na siebie wzajemnie, z których żadne nie może podjąć dalszego wykonania — nazywa się to *zakleszczeniem* (ang. *deadlock*)[21] tudzież *blokadą wzajemną*.

Jeśli podczas wykonania programu zakleszczenie występuje od razu, możesz natychmiast podjąć diagnostykę. Gorzej, jeśli program zdaje się działać poprawnie, lecz ma ukrytą zdolność powodowania zakleszczenia. W takiej sytuacji nie mamy żadnych wskazówek o możliwości wystąpienia zakleszczenia, przez co będzie ono czyhało „w uśpieniu", aż do czasu gdy nieoczekiwanie przysporzy problemów użytkownikowi (i to najpewniej w sytuacji niedającej się odtworzyć w celach diagnostycznych). Dlatego też kluczowym zagadnieniem tworzenia programów współbieżnych jest zapobieganie wzajemnym blokadom poprzez staranne zaprojektowanie programu.

Klasycznym przykładem zakleszczenia jest wymyślony przez Edsgera Dijkstrę problem *ucztujących filozofów*. W podstawowym opisie problemu występuje pięciu filozofów (choć zaprezentowany poniżej przykład pozwala na określanie ich dowolnej liczby). Filozofowie część czasu spędzają na rozmyślaniach, a część na jedzeniu obiadu. Podczas rozmyślań filozofowie nie wykorzystują żadnych zasobów współdzielonych; jednak podczas spożywania obiadu zasiadają przy stole, na którym znajduje się ograniczona liczba sztućców. W oryginalnym opisie problemu sztućcami tymi były widelce, i do jedzenia spaghetti z miski na środku stołu konieczne było zdobycie dwóch widelców. Bardziej sensowne wydaje się jednak założenie, że sztućcami tymi są pałeczki; a zatem każdy filozof, aby spożyć obiad, musi zdobyć dwie pałeczki.

Do przykładu dodano pewne utrudnienie. Otóż ze względu na swe zajęcie filozofowie mają bardzo mało pieniędzy i stać ich wyłącznie na kupno pięciu pałeczek (w ujęciu ogólnym liczba pałeczek jest równa liczbie filozofów). Pałeczki te są rozmieszczone na stole pomiędzy poszczególnymi filozofami. Gdy któryś z filozofów chce jeść, musi zdobyć pałeczkę znajdującą się z jego lewej i prawej strony. Jeśli inny filozof znajdujący się z którejkolwiek ze stron używa już pożądanej pałeczki, nasz filozof musi poczekać.

```
//: concurrency/Chopstick.java
// Pałeczki dla biesiadujących filozofów.

public class Chopstick {
  private boolean taken = false;
  public synchronized
  void take() throws InterruptedException {
```

[21] Podobna sytuacja (z ang. *livelock*) zdarza się, kiedy zadania mogą wzajemnie zmieniać swój stan (nie blokując się), ale nie prowadzi to do jakichkolwiek istotnych postępów.

```
    while(taken)
      wait();
    taken = true;
  }
  public synchronized void drop() {
    taken = false;
    notifyAll();
  }
} ///:~
```

Dwóch sąsiadujących przy stole filozofów nie może jednocześnie wziąć (`take()`) tej samej pałeczki. Do tego, jeśli pałeczka pozostaje w posiadaniu jednego z filozofów, inny może wywołać metodę `wait()` w celu zawieszenia w oczekiwaniu na dostępność pałeczki, którą jej bieżący posiadacz odłoży wywołaniem metody `drop()`.

Kiedy zadanie filozofa (`Philosopher`) wywołuje metodę `take()`, filozof jest zawieszany w oczekiwaniu, aż znacznik zajętości pałeczki (`taken`) zmieni wartość na `false` (czyli do momentu, w którym filozof przetrzymujący pałeczkę odłoży ją na miejsce). Potem zadanie ustawia znacznik `taken` na `true`, ogłaszając fakt zawłaszczenia pałeczki przez nowego filozofa. Kiedy wreszcie ten filozof zakończy posiłek, odkłada pałeczkę wywołaniem `drop()` (zmieniając znacznik zajętości) i wywołuje `notifyAll()`, wybudzając z letargu filozofów oczekujących na pałeczki.

```
//: concurrency/Philosopher.java
// Ucztujący filozof
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Philosopher implements Runnable {
  private Chopstick left;
  private Chopstick right;
  private final int id;
  private final int ponderFactor;
  private Random rand = new Random(47);
  private void pause() throws InterruptedException {
    if(ponderFactor == 0) return;
    TimeUnit.MILLISECONDS.sleep(
      rand.nextInt(ponderFactor * 250));
  }
  public Philosopher(Chopstick left, Chopstick right,
    int ident, int ponder) {
    this.left = left;
    this.right = right;
    id = ident;
    ponderFactor = ponder;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        print(this + " " + "rozmyśla");
        pause();
        // Filozof zgłodniał
        print(this + " " + "chwyta prawą pałeczkę");
        right.take();
        print(this + " " + "chwyta lewą pałeczkę");
```

```
          left.take();
          print(this + " " + "ucztuje");
          pause();
          right.drop();
          left.drop();
        }
      } catch(InterruptedException e) {
        print(this + " " + "wyjście wymuszone przerwaniem");
      }
    }
    public String toString() { return "Filozof " + id; }
} ///:~
```

W metodzie `Philosopher.run()` każdy z filozofów realizuje cykl rozmyślania i ucztowania. Jeśli pole `ponderFactor` (skłonność do rozmyślań) ma wartość niezerową, metoda `pause()` usypia zadanie filozofa na losowy czas. Filozof jakiś czas rozmyśla, a potem próbuje chwycić prawą i lewą pałeczkę, następnie jakiś czas zajada, a potem zaczyna cały cykl od początku.

Teraz możemy przygotować taką wersję programu, która doprowadzi do zakleszczenia:

```
//: concurrency/DeadlockingDiningPhilosophers.java
// Ilustruje ryzyko zaszycia zakleszczenia w programie.
// {Args: 0 5 timeout}
import java.util.concurrent.*;

public class DeadlockingDiningPhilosophers {
  public static void main(String[] args) throws Exception {
    int ponder = 5;
    if(args.length > 0)
      ponder = Integer.parseInt(args[0]);
    int size = 5;
    if(args.length > 1)
      size = Integer.parseInt(args[1]);
    ExecutorService exec = Executors.newCachedThreadPool();
    Chopstick[] sticks = new Chopstick[size];
    for(int i = 0; i < size; i++)
      sticks[i] = new Chopstick();
    for(int i = 0; i < size; i++)
      exec.execute(new Philosopher(
        sticks[i], sticks[(i+1) % size], i, ponder));
    if(args.length == 3 && args[2].equals("timeout"))
      TimeUnit.SECONDS.sleep(5);
    else {
      System.out.println("Aby zakończyć, naciśnij 'Enter'");
      System.in.read();
    }
    exec.shutdownNow();
  }
} /* (Execute to see output) *///:~
```

Łatwo zauważyć, że kiedy filozofowie mało co rozmyślają, rywalizują zaciekle o sztućce, próbując rozpocząć ucztowanie, przez co szybko dochodzi do zakleszczenia.

Pierwszy argument wiersza wywołania programu (`ponder`) ustala współczynnik skłonności do rozmyślań regulujący ilość czasu spędzanego przez filozofa na rozmyślaniu.

Jeśli filozofów jest wielu albo kiedy spędzają dużo czasu na rozmyślaniach, do zakleszczenia dochodzi rzadko, można też w ogóle go nie doczekać. Natomiast wyzerowanie skłonności do rozmyślań prowadzi do zakleszczenia stosunkowo szybko.

Zauważ, że obiekty pałeczek (`Chopstick`) nie potrzebują wewnętrznych identyfikatorów; są identyfikowane na podstawie pozycji w tablicy `sticks`. Konstruktor każdego egzemplarza klasy `Philosopher` otrzymuje referencję lewej i prawej pałeczki. Każdy z filozofów (poza ostatnim) jest inicjalizowany przez spozycjonowanie go pomiędzy następną parą pałeczek. Ostatni filozof otrzymuje jako pałeczkę zerową referencję zerowego elementu tablicy, zamykając tym samym krąg biesiadny — ostatni filozof zasiada obok pierwszego i obaj korzystają ze wspólnej pałeczki. W takim układzie istnieje możliwość, że wszyscy filozofowie będą próbować ucztowania, czekając, aż ich sąsiedzi odłożą swoje pałeczki. W ten sposób dojdzie do zakleszczenia programu.

Jeśli filozofowie spędzają więcej czasu na myśleniu, a mniej na jedzeniu, prawdopodobieństwo kolizji przy próbie pozyskania wspólnego zasobu (pałeczki) maleje, przez co program może zdawać się wolny od ryzyka zakleszczenia (zwłaszcza przy niezerowym współczynniku skłonności do rozmyślania albo wielkiej liczbie filozofów), choć w rzeczywistości jest na nie narażony. Przykład jest ciekawy, gdyż pokazuje, że program może sprawiać wrażenie całkowitej poprawności, a jednocześnie mieć skłonności do powodowania wzajemnej blokady.

Aby rozwiązać problem, należy zrozumieć, że wzajemna blokada może wystąpić, gdy jednocześnie zajdzie:

1. Wzajemne wykluczanie: przynajmniej jeden zasób wspólny musi nie nadawać się do równoczesnego wykorzystania przez wiele zadań. W tym przypadku, w danej chwili, pałeczka może być używana wyłącznie przez jednego filozofa.
2. Przynajmniej jedno zadanie musi posiadać zasób i oczekiwać na uzyskanie innego zasobu, aktualnie posiadanego przez inne zadanie. Czyli, aby wystąpiło zakleszczenie, filozof musi mieć jedną pałeczkę i czekać na drugą.
3. Nie może istnieć możliwość wywłaszczania zasobów zadaniom, które je posiadają. Wszystkie zadania muszą zwalniać zasoby w normalny sposób. Nasi filozofowie są uprzejmi i nie wydzierają sobie nawzajem pałeczek z rąk.
4. Musi nastąpić zapętlone oczekiwanie, czyli jedno zadanie musi czekać na zasób posiadany przez inne zadanie, które z kolei czeka na zasób posiadany przez jeszcze inne zadanie itd., aż w końcu ostatnie zadanie musi czekać na zasób posiadany przez pierwsze zadanie. W programie *DeadlockingDinningPhilosophers.java* zapętlone oczekiwanie pojawia się, ponieważ każdy filozof stara się najpierw zdobyć prawą pałeczkę, a potem lewą.

Ponieważ *wszystkie* te warunki muszą zostać spełnione, aby wystąpiła wzajemna blokada, wystarczy uniemożliwić zajście dowolnego z nich, aby uchronić się przed nim. W naszym przykładzie najprostszym sposobem zapobieżenia wzajemnej blokadzie jest wykluczenie czwartego warunku. Zachodzi on, ponieważ każdy z filozofów stara się zdobywać pałeczki w ściśle określonej kolejności — najpierw prawą, a potem lewą. Dlatego możliwe jest wystąpienie sytuacji, w której każdy z nich posiada prawą pałeczkę i czeka na lewą, co powoduje spełnienie warunku zapętlonego oczekiwania. Jeśli jednak ostatni filozof zostanie zainicjowany w taki sposób, by najpierw próbował zdobyć lewą pałeczkę,

a potem prawą, to nigdy nie uniemożliwi on sąsiadowi z prawej zdobycia jego kompletu pałeczek. Dzięki temu warunek zapętlonego oczekiwania nigdy nie zostanie spełniony. To tylko jeden ze sposobów rozwiązania problemu, można go także rozwiązać, wykluczając któryś z pozostałych warunków (więcej szczegółów na ten temat można znaleźć w bardziej zaawansowanych książkach poświęconych wielowątkowości).

```
//: concurrency/FixedDiningPhilosophers.java
// Filozofowie ucztujący bez ryzyka zakleszczenia.
// {Args: 5 5 timeout}
import java.util.concurrent.*;

public class FixedDiningPhilosophers {
  public static void main(String[] args) throws Exception {
    int ponder = 5;
    if(args.length > 0)
      ponder = Integer.parseInt(args[0]);
    int size = 5;
    if(args.length > 1)
      size = Integer.parseInt(args[1]);
    ExecutorService exec = Executors.newCachedThreadPool();
    Chopstick[] sticks = new Chopstick[size];
    for(int i = 0; i < size; i++)
      sticks[i] = new Chopstick();
    for(int i = 0; i < size; i++)
      if(i < (size-1))
        exec.execute(new Philosopher(
          sticks[i], sticks[i+1], i, ponder));
      else
        exec.execute(new Philosopher(
          sticks[0], sticks[i], i, ponder));
    if(args.length == 3 && args[2].equals("timeout"))
      TimeUnit.SECONDS.sleep(5);
    else {
      System.out.println("Aby zakończyć, naciśnij 'Enter'");
      System.in.read();
    }
    exec.shutdownNow();
  }
} /* (Execute to see output) *///:~
```

Zamieniając kolejność inicjalizacji w konstruktorze ostatniego obiektu `Philosopher`, przez co ostatni filozof zdobywa najpierw prawą pałeczkę, a potem lewą, można zapobiec pojawianiu się zakleszczenia.

Java jako język nie dysponuje żadnymi rozwiązaniami ułatwiającymi zapobieganie wzajemnym blokadom — to programista musi im zapobiegać, poprawnie projektując programy. Nie jest to szczególnym pocieszeniem dla osób próbujących testować wielowątkowe programy, w których zdarzają się wzajemne blokady.

Ćwiczenie 31. Zmień program *DeadlockDiningPhilosophers.java* tak, aby filozof po zakończeniu posiłku wrzucał pałeczki do ogólnodostępnego słoika, z którego filozofowie wyjmują pałeczki, kiedy zasiadają do stołu. Czy wyeliminuje to możliwość zakleszczenia? A jeśli tak, to czy można znów doprowadzić do zakleszczenia, zmniejszając liczbę pałeczek? (8).

Nowe komponenty biblioteczne

Biblioteka `java.util.concurrent` w Javie SE5 wprowadza sporą liczbę nowych klas przeznaczonych do rozwiązywania problemów charakterystycznych dla współbieżności. Ich opanowanie pozwoli Ci na projektowanie prostszych i solidniejszych programów.

Niniejszy podrozdział zawiera dość reprezentacyjny zestaw przykładów rozmaitych komponentów, ale niektóre z nich — takich, których rzadziej się używa i rzadziej spotyka — zostały pominięte.

Ponieważ omawiane komponenty służą do rozwiązywania rozmaitych problemów, nie sposób ich podzielić na rozsądne kategorie, więc będę próbował zaczynać od klas prostszych i przechodzić do klas bardziej skomplikowanych.

CountDownLatch

Klasa `CountDownLatch` (ang. *latch* — zatrzask) służy do synchronizowania jednego bądź wielu zadań przez wymuszanie oczekiwania na zakończenie zestawu operacji wykonywanych przez inne zadania.

Obiekt klasy `CountDownLatch` inicjalizuje się wartością początkową licznika, a każde zadanie wywołujące metodę `await()` na rzecz tego obiektu zostaje zablokowane, aż licznik osiągnie wartość zero. Pozostałe zadania mogą wywoływać metodę `countDown()` obiektu, która zmniejsza licznik — założenie jest takie, że zadania wywołują tę metodę po zakończeniu zadań, których się od nich oczekuje. Klasa `CountDownLatch` została zaprojektowana pod kątem zastosowania jednorazowego: licznika nie da się ponownie ustawiać. Tam, gdzie potrzebna jest wersja z możliwością regenerowania licznika, należy zastosować klasę `CyclicBarrier`.

Zadanie, które wywołuje metodę `countDown()` egzemplarza `CountDownLatch`, nie jest blokowane na wywołaniu. Blokujące jest jedynie wywołanie `await()`; wywołujące je zadanie zostanie zawieszone aż do momentu wyzerowania licznika.

Typowe zastosowanie klasy to podział problemu na *n* realizowanych niezależnie zadań i utworzenie egzemplarza `CountDownLatch` inicjalizowanego wartością *n*. Każde zadanie, które zakończy swoją pracę, wywołuje `countDown()` egzemplarza. Z kolei zadania oczekujące na rozwiązanie problemu wywołują `await()`, zawieszając się tym samym w oczekiwaniu na zakończenie wszystkich przewidzianych zadań częściowych. Oto szkielet programu ilustrującego tę technikę:

```
//: concurrency/CountDownLatchDemo.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Realizuje pewną część zadania:
class TaskPortion implements Runnable {
  private static int counter = 0;
  private final int id = counter++;
  private static Random rand = new Random(47);
```

```
  private final CountDownLatch latch;
  TaskPortion(CountDownLatch latch) {
    this.latch = latch;
  }
  public void run() {
    try {
      doWork();
      latch.countDown();
    } catch(InterruptedException ex) {
      // Dopuszczalny tryb kończenia zadania
    }
  }
  public void doWork() throws InterruptedException {
    TimeUnit.MILLISECONDS.sleep(rand.nextInt(2000));
    print(this + "zakończone");
  }
  public String toString() {
    return String.format("%1$-3d ", id);
  }
}

// Oczekiwanie na egzemplarzu CountDownLatch:
class WaitingTask implements Runnable {
  private static int counter = 0;
  private final int id = counter++;
  private final CountDownLatch latch;
  WaitingTask(CountDownLatch latch) {
    this.latch = latch;
  }
  public void run() {
    try {
      latch.await();
      print("Przejście bariery zatrzaskowej przez " + this);
    } catch(InterruptedException ex) {
      print(this + " przerwane");
    }
  }
  public String toString() {
    return String.format("zadanie WaitingTask %1$-3d ", id);
  }
}

public class CountDownLatchDemo {
  static final int SIZE = 100;
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    // Wszystkie zadania muszą współdzielić pojedynczy obiekt CountDownLatch:
    CountDownLatch latch = new CountDownLatch(SIZE);
    for(int i = 0; i < 10; i++)
      exec.execute(new WaitingTask(latch));
    for(int i = 0; i < SIZE; i++)
      exec.execute(new TaskPortion(latch));
    print("Wszystkie zadania uruchomione");
    exec.shutdown(); // Wyjście po zakończeniu wszystkich zadań
  }
} /* (Execute to see output) *///:~
```

Zadanie częściowe `TaskPortion` jest usypiane na losowy czas, co ma symulować realizację części zadania ogólnego; zadania oczekujące `WaitingTask` reprezentują te komponenty systemu, które muszą wstrzymać swoje działanie do czasu rozwiązania problemu. Wszystkie zadania korzystają ze wspólnego obiektu-bariery w postaci egzemplarza `CountDownLatch` tworzonego w metodzie `main()`.

Ćwiczenie 32. Za pomocą obiektu `CountDownLatch` rozwiąż problem korelacji odczytu rejestrów osób wchodzących do ogrodu botanicznego przez poszczególne bramki w programie *OrnamentalGarden.java*. Usuń z nowej wersji programu zbędny kod (7).

Bezpieczeństwo w środowisku wielowątkowym

Zauważ, że klasa `TaskPortion` zawiera statyczny obiekt `Random`, co oznacza potencjalną możliwość wywołania metody `Random.nextInt()` równocześnie przez wiele zadań. Czy to bezpieczne?

Gdyby okazało się to problematyczne, można by pokusić się o wyposażenie każdego obiektu `TaskPortion` we własny egzemplarz `Random`, to jest o usunięcie specyfikatora `static`. Ale pozostaje pytanie ogólniejsze, o standardowe metody Javy: które są bezpieczne, a które nie zostały zabezpieczone na okoliczność wywołań współbieżnych?

Niestety, dokumentacja JDK nic jest tu specjalnie pomocna. Akurat metoda `Random.nextInt()` nadaje się do stosowania w środowiskach wielowątkowych, ale tę cechę każdej metody trzeba ustalać samodzielnie, przeszukując fora WWW albo wprost zaglądając do kodu bibliotecznego. Nie jest to specjalnie korzystna sytuacja, zwłaszcza w kontekście języka programowania, który — przynajmniej teoretycznie — ma u zarania obsługiwać współbieżność.

CyclicBarrier

Klasa `CyclicBarrier` wykorzystywana jest w sytuacjach, w których potrzeba utworzyć grupę zadań, uruchomić je współbieżnie, a potem odczekać na ich zakończenie warunkujące przejście do następnego etapu programu (coś w rodzaju metody `join()`). Wszystkie równolegle wykonywane zadania są wyrównywane na barierze uniemożliwiającej przedwczesne kontynuowanie procesów programu. Model klasy bardzo przypomina klasę `CountDownLatch`, z tym że obiekty `CountDownLatch` to „jednorazówki", a egzemplarze `CyclicBarrier` można swobodnie regenerować.

Od samego początku mojej przygody z komputerami fascynowały mnie wszelakie symulacje, a współbieżność jest w tych fascynacjach czynnikiem kluczowym. Pierwszy program, który napisałem[22], był symulacją: symulował wyścigi konne i był napisany w języku BASIC, a zapisany pod nazwą *HOSRAC.BAS*. Oto obiektowa, wielowątkowa wersja tamtego programu ilustrująca zastosowanie klasy `CyclicBarrier`:

```
//: concurrency/HorseRace.java
// Zastosowanie klasy CyclicBarrier.
import java.util.concurrent.*;
```

[22] Jako „pierwszoroczniak" w liceum; pracownia komputerowa była wyposażona w terminal dalekopisowy ASR-33 ze 110-bodowym akustycznym modemem połączonym z komputerem HP-1000.

```
import java.util.*;
import static net.mindview.util.Print.*;

class Horse implements Runnable {
  private static int counter = 0;
  private final int id = counter++;
  private int strides = 0;
  private static Random rand = new Random(47);
  private static CyclicBarrier barrier;
  public Horse(CyclicBarrier b) { barrier = b; }
  public synchronized int getStrides() { return strides; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        synchronized(this) {
          strides += rand.nextInt(3); // Generuje wartości 0, 1 lub 2
        }
        barrier.await();
      }
    } catch(InterruptedException e) {
      // Dopuszczalny sposób opuszczenia zadania
    } catch(BrokenBarrierException e) {
      // O tym wyjątku chcielibyśmy wiedzieć
      throw new RuntimeException(e);
    }
  }
  public String toString() { return "Koń " + id + " "; }
  public String tracks() {
    StringBuilder s = new StringBuilder();
    for(int i = 0; i < getStrides(); i++)
      s.append("*");
    s.append(id);
    return s.toString();
  }
}

public class HorseRace {
  static final int FINISH_LINE = 75;
  private List<Horse> horses = new ArrayList<Horse>();
  private ExecutorService exec =
    Executors.newCachedThreadPool();
  private CyclicBarrier barrier;
  public HorseRace(int nHorses, final int pause) {
    barrier = new CyclicBarrier(nHorses, new Runnable() {
      public void run() {
        StringBuilder s = new StringBuilder();
        for(int i = 0; i < FINISH_LINE; i++)
          s.append("="); // Płot otaczający tor
        print(s);
        for(Horse horse : horses)
          print(horse.tracks());
        for(Horse horse : horses)
          if(horse.getStrides() >= FINISH_LINE) {
            print(horse + "wygrał!");
            exec.shutdownNow();
            return;
          }
```

```
        try {
          TimeUnit.MILLISECONDS.sleep(pause);
        } catch(InterruptedException e) {
          print("przerwano akcję bariery");
        }
      }
    });
    for(int i = 0; i < nHorses; i++) {
      Horse horse = new Horse(barrier);
      horses.add(horse);
      exec.execute(horse);
    }
  }
  public static void main(String[] args) {
    int nHorses = 7;
    int pause = 200;
    if(args.length > 0) { // Opcjonalny argument
      int n = new Integer(args[0]);
      nHorses = n > 0 ? n : nHorses;
    }
    if(args.length > 1) { // Opcjonalny argument
      int p = new Integer(args[1]);
      pause = p > -1 ? p : pause;
    }
    new HorseRace(nHorses, pause);
  }
} /* (Execute to see output) *///:~
```

Obiektowi `CyclicBarrier` można przekazać „akcję bariery" będącą implementacją interfejsu `Runnable` i wykonywaną automatycznie po wyzerowaniu licznika bariery. To kolejna cecha odróżniająca tę klasę od `CountDownLatch`. Akcja bariery jest tu tworzona jako anonimowa klasa przekazywana do konstruktora egzemplarza `CyclicBarrier`.

Próbowałem wypisywać postępy każdego konia osobno, ale wtedy kolejność koni na torze zależała od planisty zadań. Bariera `CyclicBarrier` pozwala każdemu wierzchowcowi na wykonanie operacji reprezentującej postęp na torze, po czym koń oczekuje na barierze dopóty, dopóki pozostałe konie również nie wykonają swoich ruchów. Kiedy wszystkie już się przesuną, bariera automatycznie wywołuje swoją akcję, która polega na wypisaniu toru z reprezentacją bieżącego położenia wierzchowców.

Kiedy wszystkie zadania przejdą barierę, jest ona automatycznie regenerowana do następnej rundy symulacji.

Aby uzyskać efekt prostej animacji wyścigu, można zmniejszyć odpowiednio okno konsoli tak, aby było na nim widać jedynie konie i tor.

DelayQueue

`DelayQueue` to nieograniczona co do liczby elementów kolejka synchronizowana, implementująca interfejs `Delayed`. Otóż obiekt może być wyjęty z kolejki tylko wtedy, kiedy upłynął czas kwarantanny. Kolejka jest sortowana tak, aby na czele kolejki znajdował się ten obiekt, którego kwarantanna upłynęła najdawniej. Jeśli żaden z obiektów

nie zakończył jeszcze kwarantanny, element czołowy kolejki nie istnieje i wywołania `poll()` zwrócą wartość `null` (z tego też powodu w kolejce nie wolno umieszczać referencji pustych).

Oto prosty przykład, w którym obiekty `Delayed` reprezentują zadania do wykonania, a zadanie `DelayedTaskConsumer` wybiera „najpilniejsze" zadanie (to, które najdawniej zakończyło kwarantannę) z kolejki i uruchamia je. Zauważ, że `DelayQueue` jest odmianą kolejki priorytetowej:

```
//: concurrency/DelayQueueDemo.java
import java.util.concurrent.*;
import java.util.*;
import static java.util.concurrent.TimeUnit.*;
import static net.mindview.util.Print.*;

class DelayedTask implements Runnable, Delayed {
  private static int counter = 0;
  private final int id = counter++;
  private final int delta;
  private final long trigger;
  protected static List<DelayedTask> sequence =
    new ArrayList<DelayedTask>();
  public DelayedTask(int delayInMilliseconds) {
    delta = delayInMilliseconds;
    trigger = System.nanoTime() +
      NANOSECONDS.convert(delta, MILLISECONDS);
    sequence.add(this);
  }
  public long getDelay(TimeUnit unit) {
    return unit.convert(
      trigger - System.nanoTime(), NANOSECONDS);
  }
  public int compareTo(Delayed arg) {
    DelayedTask that = (DelayedTask)arg;
    if(trigger < that.trigger) return -1;
    if(trigger > that.trigger) return 1;
    return 0;
  }
  public void run() { printnb(this + " "); }
  public String toString() {
    return String.format("[%1$-4d]", delta) +
      " Zadanie " + id;
  }
  public String summary() {
    return "(" + id + ":" + delta + ")";
  }
  public static class EndSentinel extends DelayedTask {
    private ExecutorService exec;
    public EndSentinel(int delay, ExecutorService e) {
      super(delay);
      exec = e;
    }
    public void run() {
      for(DelayedTask pt : sequence) {
        printnb(pt.summary() + " ");
      }
```

```
        print();
        print(this + " wywołuje shutdownNow()");
        exec.shutdownNow();
      }
    }
  }

  class DelayedTaskConsumer implements Runnable {
    private DelayQueue<DelayedTask> q;
    public DelayedTaskConsumer(DelayQueue<DelayedTask> q) {
      this.q = q;
    }
    public void run() {
      try {
        while(!Thread.interrupted())
          q.take().run(); // Uruchomienie zadania w bieżącym wątku
      } catch(InterruptedException e) {
        // Dopuszczalny sposób wychodzenia
      }
      print("Zakończono DelayedTaskConsumer");
    }
  }

  public class DelayQueueDemo {
    public static void main(String[] args) {
      Random rand = new Random(47);
      ExecutorService exec = Executors.newCachedThreadPool();
      DelayQueue<DelayedTask> queue =
        new DelayQueue<DelayedTask>();
      // Wypełnienie kolejki zadaniami o losowo
      // dobranych czasach kwarantanny:
      for(int i = 0; i < 20; i++)
        queue.put(new DelayedTask(rand.nextInt(5000)));
      // Wstawienie zadania-atrapy wymuszającego zatrzymanie
      queue.add(new DelayedTask.EndSentinel(5000, exec));
      exec.execute(new DelayedTaskConsumer(queue));
    }
  } /* Output:
  [128 ] Zadanie 11 [200 ] Zadanie 7 [429 ] Zadanie 5 [520 ] Zadanie 18 [555 ] Zadanie 1 [961 ] Zadanie 4
  [998 ] Zadanie 16 [1207] Zadanie 9 [1693] Zadanie 2 [1809] Zadanie 14 [1861] Zadanie 3 [2278]
  Zadanie 15 [3288] Zadanie 10 [3551] Zadanie 12 [4258] Zadanie 0 [4258] Zadanie 19 [4522] Zadanie 8
  [4589] Zadanie 13 [4861] Zadanie 17 [4868] Zadanie 6 (0:4258) (1:555) (2:1693) (3:1861) (4:961)
  (5:429) (6:4868) (7:200) (8:4522) (9:1207) (10:3288) (11:128) (12:3551) (13:4589) (14:1809) (15:2278)
  (16:998) (17:4861) (18:520) (19:4258) (20:5000)
  [5000] Zadanie 20 wywołuje shutdownNow()
  Zakończono DelayedTaskConsumer
  *///:~
```

Klasa `DelayedTask` zawiera kolekcję `List<DelayedTask>` o nazwie sequence, która zachowuje kolejność utworzenia zadań; dzięki niej można się przekonać, że kolejka `DelayQueue` faktycznie porządkuje obiekty na własną rękę.

Interfejs `Delayed` składa się z zaledwie jednej metody, `getDelay()`, która informuje o okresie oczekiwania do zakończenia kwarantanny albo o czasie, jaki już upłynął od jej zakończenia. Metoda ta zmusza nas do stosowania klasy `TimeUnit`, bo przyjmuje argument właśnie takiego typu. Okazuje się, że klasa `TimeUnit` jest klasą bardzo wygodną, bo ułatwia konwersję jednostek czasu bez konieczności wykonywania jakichkolwiek jawnych

obliczeń. Na przykład wartość delta jest wyrażana w milisekundach, a metoda System.nanoTime() zwraca czas w nanosekundach. Wartość delta można przeliczyć na pożądane jednostki, podając jednostkę bieżącą i jednostkę docelową, jak tutaj:

```
NANOSECONDS.convert(delta, MILISECONDS);
```

W metodzie getDelay() pożądane jednostki czasowe są przekazywane za pośrednictwem argumentu unit, co służy nam do konwersji czasu, jaki upłynął od zainicjowania odliczania na jednostki wymagane przez wywołującego, i to bez znajomości tych jednostek (to proste zastosowanie wzorca projektowego *Strategy*, gdzie część algorytmu jest przekazywana w postaci argumentu).

Co do sortowania, to interfejs Delayed dziedziczy również interfejs Comparable, co wymusza implementację metody compareTo() w celu zapewnienia odpowiedniego schematu porównania obiektów Delayed. Metody toString() i summary() służą do formatowania informacji wyjściowych, a zagnieżdżona klasa EndSentinel implementuje znacznik umożliwiający zamknięcie całego programu po wykryciu tego znacznika w kolejce.

Zauważ, że ponieważ obiekt klasy DelayedTaskConsumer sam jest zadaniem, posiada własny wątek, w którym może uruchamiać zadania wyciągane z kolejki. Ponieważ zadania są uruchamiane w kolejności zgodnej z priorytetem kolejki, nie ma potrzeby uruchamiania osobnych wątków dla poszczególnych zadań DelayedTask — wystarczy, że są uruchamiane kolejno, według ważności.

Na wyjściu programu widać, że kolejność tworzenia zadań nie ma wpływu na kolejność ich wykonywania: zadania są uruchamiane — zgodnie z oczekiwaniem — w kolejności zgodnej z wartością opóźnienia („kwarantanny").

PriorityBlockingQueue

To kolejka priorytetowa synchronizowana, z blokującymi operacjami wyciągania elementów z kolejki. Poniżej prezentuję przykład, w którym obiekty umieszczane w kolejce priorytetowej są zadaniami napływającymi do kolejki zgodnie z wartością priorytetu. Aby zapewnić odpowiedni porządek uruchamiania, zadanie PrioritizedTask posiada pole reprezentujące priorytet:

```
//: concurrency/PriorityBlockingQueueDemo.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class PrioritizedTask implements
Runnable, Comparable<PrioritizedTask> {
  private Random rand = new Random(47);
  private static int counter = 0;
  private final int id = counter++;
  private final int priority;
  protected static List<PrioritizedTask> sequence =
    new ArrayList<PrioritizedTask>();
  public PrioritizedTask(int priority) {
    this.priority = priority;
    sequence.add(this);
```

```
  }
  public int compareTo(PrioritizedTask arg) {
    return priority < arg.priority ? 1 :
      (priority > arg.priority ? -1 : 0);
  }
  public void run() {
    try {
      TimeUnit.MILLISECONDS.sleep(rand.nextInt(250));
    } catch(InterruptedException e) {
      // Dopuszczalny sposób wyjścia
    }
    print(this);
  }
  public String toString() {
    return String.format("[%1$-3d]", priority) +
      " Zadanie " + id;
  }
  public String summary() {
    return "(" + id + ":" + priority + ")";
  }
  public static class EndSentinel extends PrioritizedTask {
    private ExecutorService exec;
    public EndSentinel(ExecutorService e) {
      super(-1); // Najniższy możliwy priorytet w programie
      exec = e;
    }
    public void run() {
      int count = 0;
      for(PrioritizedTask pt : sequence) {
        printnb(pt.summary());
        if(++count % 5 == 0)
          print();
      }
      print();
      print(this + " wywołało shutdownNow()");
      exec.shutdownNow();
    }
  }
}

class PrioritizedTaskProducer implements Runnable {
  private Random rand = new Random(47);
  private Queue<Runnable> queue;
  private ExecutorService exec;
  public PrioritizedTaskProducer(
    Queue<Runnable> q, ExecutorService e) {
    queue = q;
    exec = e; // Dla obiektu EndSentinel
  }
  public void run() {
    // Kolejka nieograniczona; nie blokuje.
    // Szybkie wypełnianie zadaniami z losowymi priorytetami:
    for(int i = 0; i < 20; i++) {
      queue.add(new PrioritizedTask(rand.nextInt(10)));
      Thread.yield();
    }
    // Jeszcze kilka zadań o najwyższym priorytecie:
    try {
```

```
      for(int i = 0; i < 10; i++) {
        TimeUnit.MILLISECONDS.sleep(250);
        queue.add(new PrioritizedTask(10));
      }
      // Dodawanie zadań, od najniższego priorytetu:
      for(int i = 0; i < 10; i++)
        queue.add(new PrioritizedTask(i));
      // Zadanie-atrapa zatrzymujące wszystkie zadania:
      queue.add(new PrioritizedTask.EndSentinel(exec));
    } catch(InterruptedException e) {
      // Dopuszczalny sposób wyjścia
    }
    print("Zakończono PrioritizedTaskProducer");
  }
}

class PrioritizedTaskConsumer implements Runnable {
  private PriorityBlockingQueue<Runnable> q;
  public PrioritizedTaskConsumer(
    PriorityBlockingQueue<Runnable> q) {
    this.q = q;
  }
  public void run() {
    try {
      while(!Thread.interrupted())
        // Uruchomienie zadania w bieżącym wątku:
        q.take().run();
    } catch(InterruptedException e) {
      // Dopuszczalny sposób wyjścia
    }
    print("Zakończono PrioritizedTaskConsumer");
  }
}

public class PriorityBlockingQueueDemo {
  public static void main(String[] args) throws Exception {
    Random rand = new Random(47);
    ExecutorService exec = Executors.newCachedThreadPool();
    PriorityBlockingQueue<Runnable> queue =
      new PriorityBlockingQueue<Runnable>();
    exec.execute(new PrioritizedTaskProducer(queue, exec));
    exec.execute(new PrioritizedTaskConsumer(queue));
  }
} /* (Execute to see output) *///:~
```

Tak jak w poprzednim przykładzie, sekwencja tworzenia obiektów `PrioritizedTask` jest zapamiętywana w obiekcie `sequence`, dzięki czemu można ją porównać z kolejnością faktycznego wykonywania zadań. Metoda `run()` usypia zadanie na losowo określony czas, a potem wypisuje informacje o obiekcie. Wyróżnione zadanie `EndSentinel`, jako ostatnie w kolejce, działa tak samo jak w poprzednich przykładach.

Zadania `PrioritizedTaskProducer` i `PrioritizedTaskConsumer` nawiązują połączenie za pośrednictwem priorytetowej kolejki synchronizowanej `PriorityBlockingQueue`. Blokująca natura kolejki gwarantuje konieczną w takich przypadkach synchronizację; nie trzeba implementować żadnych jawnych mechanizmów synchronizacji i nie trzeba się

przy odczycie elementów zastanawiać, czy w ogóle kolejka zawiera jakieś elementy — jeśli ich tam nie będzie, kolejka najzwyczajniej zablokuje zadanie odczytujące do czasu dostępności nowych elementów.

Sterowanie szklarnią — planowanie uruchamiania zadań

W rozdziale „Klasy wewnętrzne" występował przykład obrazujący działanie systemu sterowniczego hipotetycznej szklarni, który włączał i wyłączał znajdujące się w niej urządzenia. Taki model również można przedstawić jako problem współbieżności, gdzie każde zdarzenie dotyczące pracy szklarni jest zadaniem uruchamianym o określonym z góry czasie. Infrastrukturą niezbędną do rozwiązania tego problemu dysponuje klasa `ScheduledThreadPoolExecutor`. Za pomocą metody `schedule()` (jednokrotne uruchomienie zadania) albo `scheduleAtFixedRate()` (uruchamianie zadania w regularnych odstępach czasu) tej klasy można zaplanować uruchamianie obiektów `Runnable`. Porównaj to rozwiązanie z rozwiązaniem prezentowanym w rozdziale „Klasy wewnętrzne"; zwróć uwagę na to, jak mocno uprościło się rozwiązanie dzięki zastosowaniu gotowego narzędzia w postaci `ScheduledThreadPoolExecutor`:

```
// : concurrency/GreenhouseScheduler.java
// Reimplementacja przykładu innerclasses/GreenhouseController.java
// z użyciem wykonawcy ScheduledThreadPoolExecutor.
// {Args: 5000}
import java.util.concurrent.*;
import java.util.*;

public class GreenhouseScheduler {
  private volatile boolean light = false;
  private volatile boolean water = false;
  private String thermostat = "Dzień";
  public synchronized String getThermostat() {
    return thermostat;
  }
  public synchronized void setThermostat(String value) {
    thermostat = value;
  }
  ScheduledThreadPoolExecutor scheduler =
    new ScheduledThreadPoolExecutor(10);
  public void schedule(Runnable event, long delay) {
    scheduler.schedule(event,delay,TimeUnit.MILLISECONDS);
  }
  public void
  repeat(Runnable event, long initialDelay, long period) {
    scheduler.scheduleAtFixedRate(
      event, initialDelay, period, TimeUnit.MILLISECONDS);
  }
  class LightOn implements Runnable {
    public void run() {
      // Tu kod sterujący urządzeniami
      // uruchamiającymi oświetlenie.
      System.out.println("Włączanie świateł");
      light = true;
    }
  }
```

```
class LightOff implements Runnable {
  public void run() {
    // Tu kod sterujący urządzeniami
    // uruchamiającymi oświetlenie.
    System.out.println("Wyłączanie świateł");
    light = false;
  }
}
class WaterOn implements Runnable {
  public void run() {
    // Tu kod sterujący urządzeniami
    System.out.println("Obieg wody w szklarni włączony");
    water = true;
  }
}
class WaterOff implements Runnable {
  public void run() {
    // Tu kod sterujący urządzeniami
    System.out.println("Obieg wody w szklarni wyłączony");
    water = false;
  }
}
class ThermostatNight implements Runnable {
  public void run() {
    // Tu kod sterujący urządzeniami
    System.out.println("Termostat przełączony na ustawienie nocne");
    setThermostat("Noc");
  }
}
class ThermostatDay implements Runnable {
  public void run() {
    // Tu kod sterujący urządzeniami
    System.out.println("Termostat przełączony na ustawienie dzienne");
    setThermostat("Dzień");
  }
}
class Bell implements Runnable {
  public void run() { System.out.println("Dzyń!"); }
}
class Terminate implements Runnable {
  public void run() {
    System.out.println("Wyłączanie systemu");
    scheduler.shutdownNow();
    // Trzeba uruchomić osobne zadanie,
    // bo wykonawca-planista został zamknięty:
    new Thread() {
      public void run() {
        for(DataPoint d : data)
          System.out.println(d);
      }
    }.start();
  }
}
// Nowy element: zbieranie danych
static class DataPoint {
  final Calendar time;
  final float temperature;
  final float humidity;
```

```
    public DataPoint(Calendar d, float temp, float hum) {
      time = d;
      temperature = temp;
      humidity = hum;
    }
    public String toString() {
      return time.getTime() +
        String.format(
          " temperatura: %1$.1f wilgotność: %2$.2f",
          temperature, humidity);
    }
  }
  private Calendar lastTime = Calendar.getInstance();
  { // Wyrównanie daty do pół godziny
    lastTime.set(Calendar.MINUTE, 30);
    lastTime.set(Calendar.SECOND, 00);
  }
  private float lastTemp = 18.3f;
  private int tempDirection = +1;
  private float lastHumidity = 50.0f;
  private int humidityDirection = +1;
  private Random rand = new Random(47);
  List<DataPoint> data = Collections.synchronizedList(
    new ArrayList<DataPoint>());
  class CollectData implements Runnable {
    public void run() {
      System.out.println("Zbieranie danych");
      synchronized(GreenhouseScheduler.this) {
        // Udajemy, że interwał jest dłuższy niż faktyczny:
        lastTime.set(Calendar.MINUTE,
          lastTime.get(Calendar.MINUTE) + 30);
        // Raz na pięć prób odwracamy trend:
        if(rand.nextInt(5) == 4)
          tempDirection = -tempDirection;
        // Zachowanie poprzedniej wartości:
        lastTemp = lastTemp +
          tempDirection * (0.5f + rand.nextFloat());
        if(rand.nextInt(5) == 4)
          humidityDirection = -humidityDirection;
        lastHumidity = lastHumidity +
          humidityDirection * rand.nextFloat();
        // Obiekt Calendar musi zostać sklonowany, inaczej
        // wszystkie obiekty danych DataPoints będą zawierały
        // referencję tego samego momentu w czasie. W przypadku
        // prostych obiektów, jak Calendar, wystarczy metoda clone().
        data.add(new DataPoint((Calendar)lastTime.clone(),
          lastTemp, lastHumidity));
      }
    }
  }
  public static void main(String[] args) {
    GreenhouseScheduler gh = new GreenhouseScheduler();
    gh.schedule(gh.new Terminate(), 5000);
    // Wcześniejsza klasa "Restart" już zbędna:
    gh.repeat(gh.new Bell(), 0, 1000);
    gh.repeat(gh.new ThermostatNight(), 0, 2000);
    gh.repeat(gh.new LightOn(), 0, 200);
```

```
    gh.repeat(gh.new LightOff(), 0, 400);
    gh.repeat(gh.new WaterOn(), 0, 600);
    gh.repeat(gh.new WaterOff(), 0, 800);
    gh.repeat(gh.new ThermostatDay(), 0, 1400);
    gh.repeat(gh.new CollectData(), 500, 500);
  }
} /* (Execute to see output) *///:~
```

Mamy tu do czynienia z dalece idącą reorganizacją kodu i jednym nowym elementem: akwizycją danych o temperaturze i wilgotności w szklarni. Do przechowywania i wypisywania danych z poszczególnych momentów służą obiekty `DataPoint`, a zbiera je zadanie `CollectData` — zaplanowane zadanie generujące symulowane dane i dodające je do kontenera `List<DataPoint>`.

Zauważ, że w odpowiednich miejscach kodu pojawiły się specyfikatory `synchronized` i `volatile` zapobiegające kolizjom zadań. Wszelkie metody kontenera `List`, który przechowuje obiekty danych `DataPoint`, są synchronizowane, o co dba wywołanie `synchronizedList()` (z biblioteki `java.util.Collections`) otaczające wywołanie konstruktora właściwej listy.

Ćwiczenie 33. Zmodyfikuj program *GreenhouseScheduler.java* tak, aby zamiast wykonawcy `ScheduledExecutor` korzystał z kolejki `DelayQueue` (7).

Semaphore

Zwykłe blokady (czy to w postaci wbudowanej — `synchronized`, czy w postaci obiektów jawnych z biblioteki `concurrent.locks`) pozwalają na korzystanie ze współdzielonego zasobu w danym momencie tylko jednemu zadaniu. Tymczasem *semafor zliczający* (ang. *counting semaphore*) pozwala korzystać z zasobu wspólnego n zadaniom. Semafor jest więc czymś w rodzaju mechanizmu przydziału „zezwoleń" na użycie zasobu, mimo braku obiektów reprezentujących same „zezwolenia".

W ramach przykładu przyjrzymy się koncepcji *puli obiektów* zarządzającej pewną liczbą obiektów, które można wyciągać z puli z przeznaczeniem do jakichś operacji, a potem oddawać do puli. Tego rodzaju mechanizm można by zamknąć w klasie uogólnionej:

```
//: concurrency/Pool.java
// Zastosowanie do obsługi puli obiektów semafora
// ograniczającego liczbę zadań używających zasobów.
import java.util.concurrent.*;
import java.util.*;

public class Pool<T> {
  private int size;
  private List<T> items = new ArrayList<T>();
  private volatile boolean[] checkedOut;
  private Semaphore available;
  public Pool(Class<T> classObject, int size) {
    this.size = size;
    checkedOut = new boolean[size];
    available = new Semaphore(size, true);
    // Wypełnienie puli obiektami, które można z niej wyciągać:
    for(int i = 0; i < size; ++i)
```

```
      try {
        // Zakładamy obecność konstruktora domyślnego:
        items.add(classObject.newInstance());
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
    }
    public T checkOut() throws InterruptedException {
      available.acquire();
      return getItem();
    }
    public void checkIn(T x) {
      if(releaseItem(x))
        available.release();
    }
    private synchronized T getItem() {
      for(int i = 0; i < size; ++i)
        if(!checkedOut[i]) {
          checkedOut[i] = true;
          return items.get(i);
        }
      return null; // Semafor zapobiega dojściu tutaj
    }
    private synchronized boolean releaseItem(T item) {
      int index = items.indexOf(item);
      if(index == -1) return false; // Brak na liście
      if(checkedOut[index]) {
        checkedOut[index] = false;
        return true;
      }
      return false; // Obiekt spoza puli
    }
} ///:~
```

W tej uproszczonej postaci konstruktor puli wypełnia ją wywołaniami metody `newInstance()`. Jeśli użytkownik potrzebuje obiektu, wyjmuje go z puli wywołaniem `checkOut()`, a kiedy zakończy używanie obiektu, odkłada go do puli wywołaniem `checkIn()`.

Tablica wartości logicznych (`boolean`) o nazwie `checkedOut` reprezentuje stan puli, zaznaczając obiekty wyciągane i dostępne w puli. Tablica jest zarządzana wywołaniami `getItem()` i `releaseItem()`. Te z kolei są zabezpieczane zmienną semaforową `available`, tak że w metodzie `checkOut()` semafor `available` blokuje postęp wywołania, jeśli wyczerpią się „zezwolenia" (co oznacza w tym przypadku wyczerpanie puli obiektów). W metodzie `checkIn()`, jeśli tylko odkładany obiekt jest poprawny (to znaczy, jeśli pochodzi z puli), semafor jest uzupełniany jednym „zezwoleniem".

Do dokończenia przykładu użyjemy klasy `Fat`, czyli klasy obiektów o kosztownym czasowo procesie kreacji, co uzasadnia utworzenie puli takich obiektów:

```
//: concurrency/Fat.java
// Obiekty o czasochłonnej konstrukcji.

public class Fat {
  private volatile double d; // Blokada optymalizacji
  private static int counter = 0;
  private final int id = counter++;
```

```
  public Fat() {
    // Kosztowna czasowo operacja konstrukcji:
    for(int i = 1; i < 10000; i++) {
      d += (Math.PI + Math.E) / (double)i;
    }
  }
  public void operation() { System.out.println(this); }
  public String toString() { return "Obiekt Fat id: " + id; }
} ///:~
```

Aby ograniczyć negatywny wpływ konstruktora tej klasy na wydajność programu, decydujemy się na utworzenie ograniczonej puli obiektów Fat. Teraz możemy przetestować klasę puli Pool, tworząc zadanie wybierające z niej obiekty Fat, przetrzymujące je przez chwilę i oddające do puli:

```
//: concurrency/SemaphoreDemo.java
// Test klasy Pool
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Zadanie wyjmujące zasoby z puli:
class CheckoutTask<T> implements Runnable {
  private static int counter = 0;
  private final int id = counter++;
  private Pool<T> pool;
  public CheckoutTask(Pool<T> pool) {
    this.pool = pool;
  }
  public void run() {
    try {
      T item = pool.checkOut();
      print(this + "wyjmuje " + item);
      TimeUnit.SECONDS.sleep(1);
      print(this +"oddaje " + item);
      pool.checkIn(item);
    } catch(InterruptedException e) {
      // Dopuszczalny sposób przerwania zadania
    }
  }
  public String toString() {
    return "Zadanie CheckoutTask " + id + " ";
  }
}

public class SemaphoreDemo {
  final static int SIZE = 25;
  public static void main(String[] args) throws Exception {
    final Pool<Fat> pool =
      new Pool<Fat>(Fat.class, SIZE);
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < SIZE; i++)
      exec.execute(new CheckoutTask<Fat>(pool));
    print("Utworzono komplet zadań CheckoutTasks");
    List<Fat> list = new ArrayList<Fat>();
    for(int i = 0; i < SIZE; i++) {
      Fat f = pool.checkOut();
```

```
        printnb(i + ": wątek metody main() wyjmuje z puli ");
        f.operation();
        list.add(f);
      }
      Future<?> blocked = exec.submit(new Runnable() {
        public void run() {
          try {
            // Semafor zapobiega dalszemu wyjmowaniu,
            // blokując wywołanie:
            pool.checkOut();
          } catch(InterruptedException e) {
            print("Przerwanie w metodzie checkOut()");
          }
        }
      });
      TimeUnit.SECONDS.sleep(2);
      blocked.cancel(true); // Wyłamanie z zablokowanego wywołania
      print("Oddawanie obiektów " + list);
      for(Fat f : list)
        pool.checkIn(f);
      for(Fat f : list)
        pool.checkIn(f); // Drugie wywołanie checkIn() ignorowane
      exec.shutdown();
    }
  } /* (Execute to see output) *///:~
```

W metodzie `main()` tworzona jest pula obiektów `Fat` oraz zestaw zadań `CheckoutTask` korzystających z puli. Konkuruje z nimi sam wątek metody `main()`, który również wyjmuje obiekty `Fat` z puli, zawłaszczając je (*to jest — nie oddając do puli*). Kiedy metoda `main()` wyjmie już wszystkie obiekty z puli, następne wyjęcie zostanie zablokowane na semaforze. Metoda `run()` obiektu `blocked` zostaje wtedy zablokowana, a po dwóch sekundach zostaje odwołana przez wywołanie metody `cancel()`. Zauważ, że nadmiarowe żądania wyjęcia są przez pulę ignorowane.

Przykład ten bazuje na staranności i solidności programistów-klientów klasy `Pool`, zakładając, że prędzej czy później każdy wyjęty z puli obiekt trafi do niej z powrotem). Rozwinięcie sposobów zarządzania obiektami wyjmowanymi z pul można znaleźć w książce *Thinking in Patterns* (dostępnej w witrynie *www.MindView.net*).

Exchanger

Klasa `Exchanger` implementuje barierę, która wymienia obiekty pomiędzy dwoma zadaniami. Kiedy zadanie dociera do takiej bariery, posiada pewien obiekt, a kiedy opuszcza barierę, jego miejsce zajmuje inny obiekt, wcześniej będący w posiadaniu innego zadania. Bariery wymiany są wykorzystywane, kiedy jedno z zadań wytwarza obiekty kosztowne w produkcji, a inne zadanie konsumuje te obiekty; typowo dochodzi wtedy niejako do recyclingu obiektów, co zmniejsza koszt wytwarzania.

Aby przećwiczyć stosowanie klasy `Exchanger`, utworzymy zadania producenta i konsumenta, które za pośrednictwem typów ogólnych i generatorów będą mogły operować na obiektach dowolnych typów; w taką infrastrukturę wdrożymy potem obiekty klasy `Fat`. Klasy `ExchangerProducer` i `ExchangerConsumer` będą w roli wymienianego obiektu używały kontenera `List<T>`; każda z nich zawiera też jedną barierę wymiany dla tegoż

kontenera. Wywołanie metody Exchanger.exchange() zablokuje wywołującego do czasu, aż współpracujące zadanie wywoła swoją metodę exchange(), a kiedy oba wywołania exchange()się zakończą, oba zadania przekażą sobie kontenery List<T>:

```
//: concurrency/ExchangerDemo.java
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

class ExchangerProducer<T> implements Runnable {
  private Generator<T> generator;
  private Exchanger<List<T>> exchanger;
  private List<T> holder;
  ExchangerProducer(Exchanger<List<T>> exchg,
  Generator<T> gen, List<T> holder) {
    exchanger = exchg;
    generator = gen;
    this.holder = holder;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        for(int i = 0; i < ExchangerDemo.size; i++)
          holder.add(generator.next());
        // Wymiana pełnego na pusty:
        holder = exchanger.exchange(holder);
      }
    } catch(InterruptedException e) {
      // Dopuszczalny sposób przerwania.
    }
  }
}

class ExchangerConsumer<T> implements Runnable {
  private Exchanger<List<T>> exchanger;
  private List<T> holder;
  private volatile T value;
  ExchangerConsumer(Exchanger<List<T>> ex, List<T> holder){
    exchanger = ex;
    this.holder = holder;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        holder = exchanger.exchange(holder);
        for(T x : holder) {
          value = x; // Wyciągnięcie wartości
          holder.remove(x); // W porządku dla CopyOnWriteArrayList
        }
      }
    } catch(InterruptedException e) {
      // Dopuszczalny sposób przerwania zadania.
    }
    System.out.println("Wartość końcowa: " + value);
  }
}
```

```
public class ExchangerDemo {
  static int size = 10;
  static int delay = 5; // W sekundach
  public static void main(String[] args) throws Exception {
    if(args.length > 0)
      size = new Integer(args[0]);
    if(args.length > 1)
      delay = new Integer(args[1]);
    ExecutorService exec = Executors.newCachedThreadPool();
    Exchanger<List<Fat>> xc = new Exchanger<List<Fat>>();
    List<Fat>
      producerList = new CopyOnWriteArrayList<Fat>(),
      consumerList = new CopyOnWriteArrayList<Fat>();
    exec.execute(new ExchangerProducer<Fat>(xc,
      BasicGenerator.create(Fat.class), producerList));
    exec.execute(
      new ExchangerConsumer<Fat>(xc,consumerList));
    TimeUnit.SECONDS.sleep(delay);
    exec.shutdownNow();
  }
} /* Output: (Sample)
Wartość końcowa: Obiekt Fat id: 29999
*///:~
```

W metodzie `main()` dochodzi do utworzenia wspólnego egzemplarza bariery `Exchanger` dla obu zadań i dwóch kontenerów `CopyOnWriteArrayList` będących przedmiotem wymiany. Ten konkretny wariant klasy `List` toleruje wywołania metody `remove()` w czasie przeglądania listy, bez zgłaszania wyjątku `ConcurrentModdificationException`. Zadanie `ExchangeProducer` wypełnia kontener, a potem wymienia ów pełny kontener na pusty, odebrany od zadania `ExchangerConsumer`. Z racji obecności bariery wymiany wypełnianie kontenera z jednej strony i jego opróżnianie z drugiej mogą się odbywać współbieżnie.

Ćwiczenie 34. Zmodyfikuj program *ExchangerDemo.java*, aby zamiast klasy `Fat` wykorzystywał Twoją własną klasę (1).

Symulacje

Jednym z najciekawszych i najbardziej ekscytujących zastosowań współbieżności jest modelowanie wszelkiego rodzaju procesów w postaci symulacji. Dzięki współbieżności wykonania każdy z komponentów symulacji może być implementowany jako osobne zadanie, co znakomicie ułatwia programowanie symulacji. Wiele gier i generowanych komputerowo animacji, które możemy podziwiać w filmach, to właśnie symulacje; za symulację można uznać również prezentowane wcześniej przykłady *HorseRace.java* i *GreenhouseScheduler.java*.

Symulacja okienka kasowego

Klasyczna symulacja może reprezentować dowolną sytuację, w której obiekty pojawiają się w losowych odstępach czasu i wymagają obsługi, również zajmującej losową ilość czasu, a liczba obsługujących jest ograniczona. Symulacja pozwala wtedy na określenie optymalnej liczby owych obsługujących.

W niniejszym przykładzie weźmiemy na warsztat salę obsługi klientów w banku. Każdy klient wymaga obsługi przez pewną ilość czasu, który spędzi przy okienku kasowym. Czasochłonność obsługi będzie ustalana indywidualnie dla poszczególnych klientów przez losowanie odpowiedniego współczynnika. Dodatkowo nie będzie z góry wiadomo, ilu klientów pojawi się w sali w danym okresie — również ten parametr będzie losowany.

```
//: concurrency/BankTellerSimulation.java
// Kolejki i wielowątkowość.
// {Args: 5}
import java.util.concurrent.*;
import java.util.*;

// Obiekty niemodyfikowalne, a więc niewymagające synchronizacji:
class Customer {
  private final int serviceTime;
  public Customer(int tm) { serviceTime = tm; }
  public int getServiceTime() { return serviceTime; }
  public String toString() {
    return "[" + serviceTime + "]";
  }
}

// Klasa kolejki klientów; niech wypisuje się sama na wyjściu:
class CustomerLine extends ArrayBlockingQueue<Customer> {
  public CustomerLine(int maxLineSize) {
    super(maxLineSize);
  }
  public String toString() {
    if(this.size() == 0)
      return "[Pusto]";
    StringBuilder result = new StringBuilder();
    for(Customer customer : this)
      result.append(customer);
    return result.toString();
  }
}

// Losowe dodawanie klientów do kolejki:
class CustomerGenerator implements Runnable {
  private CustomerLine customers;
  private static Random rand = new Random(47);
  public CustomerGenerator(CustomerLine cq) {
    customers = cq;
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        TimeUnit.MILLISECONDS.sleep(rand.nextInt(300));
        customers.put(new Customer(rand.nextInt(1000)));
      }
    } catch(InterruptedException e) {
      System.out.println("Przerwano zadanie CustomerGenerator");
    }
    System.out.println("Zakończono zadanie CustomerGenerator");
  }
}
```

```
class Teller implements Runnable, Comparable<Teller> {
  private static int counter = 0;
  private final int id = counter++;
  // Liczba klientów obsłużonych w czasie zmiany:
  private int customersServed = 0;
  private CustomerLine customers;
  private boolean servingCustomerLine = true;
  public Teller(CustomerLine cq) { customers = cq; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        Customer customer = customers.take();
        TimeUnit.MILLISECONDS.sleep(
          customer.getServiceTime());
        synchronized(this) {
          customersServed++;
          while(!servingCustomerLine)
            wait();
        }
      }
    } catch(InterruptedException e) {
      System.out.println("Przerwano zadanie " + this);
    }
    System.out.println("Zakończono zadanie " + this);
  }
  public synchronized void doSomethingElse() {
    customersServed = 0;
    servingCustomerLine = false;
  }
  public synchronized void serveCustomerLine() {
    assert !servingCustomerLine:"już obsługuje: " + this;
    servingCustomerLine = true;
    notifyAll();
  }
  public String toString() { return "Kasjer (Teller) " + id + " "; }
  public String shortString() { return "K" + id; }
  // Na użytek kolejki priorytetowej:
  public synchronized int compareTo(Teller other) {
    return customersServed < other.customersServed ? -1 :
      (customersServed == other.customersServed ? 0 : 1);
  }
}

class TellerManager implements Runnable {
  private ExecutorService exec;
  private CustomerLine customers;
  private PriorityQueue<Teller> workingTellers =
    new PriorityQueue<Teller>();
  private Queue<Teller> tellersDoingOtherThings =
    new LinkedList<Teller>();
  private int adjustmentPeriod;
  private static Random rand = new Random(47);
  public TellerManager(ExecutorService e,
    CustomerLine customers, int adjustmentPeriod) {
    exec = e;
    this.customers = customers;
    this.adjustmentPeriod = adjustmentPeriod;
    // Zaczynamy od pojedynczego kasjera:
```

```
    Teller teller = new Teller(customers);
    exec.execute(teller);
    workingTellers.add(teller);
  }
  public void adjustTellerNumber() {
    // To właściwy system sterowania. Przez dostosowywanie parametrów
    // można ujawnić niestabilności mechanizmu kontrolnego.
    // Jeśli kolejka jest zbyt długa, otwieramy następne okienko kasowe:
    if(customers.size() / workingTellers.size() > 2) {
      // Kiedy kasjerzy mają przerwę, albo są zajęci czymś
      // innym, przywołujemy jednego z nich z zaplecza sali:
      if(tellersDoingOtherThings.size() > 0) {
        Teller teller = tellersDoingOtherThings.remove();
        teller.serveCustomerLine();
        workingTellers.offer(teller);
        return;
      }
      // Albo zatrudniamy (konstruujemy) nowego
      Teller teller = new Teller(customers);
      exec.execute(teller);
      workingTellers.add(teller);
      return;
    }
    // Kiedy kolejka się skróci, zamykamy okienko:
    if(workingTellers.size() > 1 &&
      customers.size() / workingTellers.size() < 2)
        reassignOneTeller();
    // Kiedy kolejka zniknie, wystarczy jeden kasjer:
    if(customers.size() == 0)
      while(workingTellers.size() > 1)
        reassignOneTeller();
  }
  // Powierzenie kasjerowi innej pracy albo zwolnienie na przerwę:
  private void reassignOneTeller() {
    Teller teller = workingTellers.poll();
    teller.doSomethingElse();
    tellersDoingOtherThings.offer(teller);
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        TimeUnit.MILLISECONDS.sleep(adjustmentPeriod);
        adjustTellerNumber();
        System.out.print(customers + " { ");
        for(Teller teller : workingTellers)
          System.out.print(teller.shortString() + " ");
        System.out.println("}");
      }
    } catch(InterruptedException e) {
      System.out.println("Przerwano zadanie " + this);
    }
    System.out.println("Zakończono zadanie " + this);
  }
  public String toString() { return "Zadanie TellerManager "; }
}

public class BankTellerSimulation {
  static final int MAX_LINE_SIZE = 50;
```

```
  static final int ADJUSTMENT_PERIOD = 1000;
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    // Jeśli kolejka jest zbyt długa, klienci się zniechęcają:
    CustomerLine customers =
      new CustomerLine(MAX_LINE_SIZE);
    exec.execute(new CustomerGenerator(customers));
    // Kierownik sali przyjmuje i odwołuje kasjerów w miarę potrzeb:
    exec.execute(new TellerManager(
      exec, customers, ADJUSTMENT_PERIOD));
    if(args.length > 0) // Argument opcjonalny
      TimeUnit.SECONDS.sleep(new Integer(args[0]));
    else {
      System.out.println("Aby zakończyć, naciśnij 'Enter'");
      System.in.read();
    }
    exec.shutdownNow();
  }
} /* Output: (Sample)
[429][200][207] { K0 K1 }
[861][258][140][322] { K0 K1 }
[575][342][804][826][896][984] { K0 K1 K2 }
[984][810][141][12][689][992][976][368][395][354] { K0 K1 K2 K3 }
Przerwano zadanie Kasjer (Teller) 2
Zakończono zadanie Kasjer (Teller) 2
Przerwano zadanie Kasjer (Teller) 1
Zakończono zadanie Kasjer (Teller) 1
Przerwano zadanie Zadanie TellerManager
Zakończono zadanie Zadanie TellerManager
Przerwano zadanie Kasjer (Teller) 3
Zakończono zadanie Kasjer (Teller) 3
Przerwano zadanie Kasjer (Teller) 0
Zakończono zadanie Kasjer (Teller) 0
Przerwano zadanie CustomerGenerator
Zakończono zadanie CustomerGenerator
*///:~
```

Obiekty klientów (`Customer`) są bardzo proste — zawierają tylko jedno pole finalne typu `int`. Ponieważ stan takich obiektów nie może się zmieniać, jako obiekty niemodyfikowalne są wolne od kwestii synchronizacji i widoczności. Zadanie kasjera (`Teller`) ogranicza się do wyjmowania kolejnych obiektów `Customer` z kolejki wejściowej i przetrzymywania tych obiektów na czas „obsługi" — więc i tak każdy obiekt `Customer` będzie w danym czasie manipulowany wyłącznie z poziomu jednego zadania.

Klasa `CustomerLine` reprezentuje pojedynczą kolejkę klientów czekających na obsługę przed okienkiem kasowym konkretnego kasjera. Właściwa kolejka to `ArrayBlockingQueue`, z metodą `toString()` wypisującą zawartość kolejki.

Z kolejką kojarzona jest klasa generatora klientów `CustomerGenerator`, która w losowych odstępach czasu dodaje obiekty `Customer` do kolejki.

Zadanie `Teller` wyjmuje obiekt `Customer` z kolejki `CustomerLine` i przetwarza obiekt przez pewien czas, zliczając przy okazji liczbę klientów obsłużonych w czasie danej zmiany. Kasjer może zostać oddelegowany z okienka do innych zadań na zapleczu (`doSomethingElse()`) — nie może wtedy obsługiwać następnych klientów; kasjer może też zostać przywołany z powrotem na salę obsługi wywołaniem `serveCustomerLine()`,

kiedy liczba klientów będzie to uzasadniała. Wybór kasjera, który ma pofatygować się do okienka, odbywa się na podstawie liczby obsłużonych klientów porównywanej za pośrednictwem metody `compareTo()` — kolejka `PriorityQueue` automatycznie wysuwa najmniej zapracowanych kasjerów na front obsługi.

Sterowanie przebiegiem symulacji koncentruje się w klasie kierownika sali — `TellerManager`. On zarządza wszystkimi kasjerami i obserwuje zachowanie kolejek. Jedną z ciekawostek symulacji jest próba adaptowania działania sali w celu określenia optymalnej liczby działających kasjerów dla obserwowanego przepływu klientów. Odbywa się to w metodzie `adjustTellerNumber()`, stanowiącej system sterowania dodający i usuwający obiekty kasjerów. Wszystkie systemy kontrolne trapi problem stabilności: zbyt szybkie reakcje oznaczają niestabilność systemu, a reakcje zbyt wolne doprowadzają system do któregoś z ekstremów.

Ćwiczenie 35. Zmodyfikuj program *BankTellerSimulation.java* tak, aby reprezentował klientów WWW przysyłających żądania do ustalonej liczby serwerów. Celem symulacji ma być określenie obciążenia, jakie może znieść taka grupa serwerów (8).

Symulacja sali restauracyjnej

Symulacja ta rozbudowuje program *Restaurant.java*, wprowadzając do niego większą liczbę komponentów symulacji, takich jak zamówienia (`Order`) i zastawa (`Plate`); wykorzystuje też klasy menu z rozdziału „Typy wyliczeniowe".

Nowa wersja wprowadza też kolejkę `SynchronousQueue` (nowość w Javie SE5), czyli kolejkę synchronizowaną bez pojemności wewnętrznej, przez co każde wywołanie metody `put()` umieszczające obiekt w kolejce musi być parowane wywołaniem `take()` opróżniającym kolejkę. Korzystanie z tej kolejki przypomina bezpośrednie przekazywanie rzeczy pomiędzy osobami — nie ma w pobliżu żadnego stołu, na którym można by położyć pakunek — wręczający musi poczekać, aż odbierający będzie miał wolne ręce. W omawianym przykładzie kolejka `SynchronousQueue` będzie reprezentować zastawę — trudno na jednej zastawie położyć więcej, niż jedno danie.

Reszta klas i zadań naśladuje strukturę programu *Restaurant.java*, ewentualnie naśladuje dość dokładnie procesy odbywające się w rzeczywistej restauracji:

```
//: concurrency/restaurant2/RestaurantWithQueues.java
// {Args: 5}
package concurrency.restaurant2;
import enumerated.menu.*;
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Obiekt wręczany kelnerowi, który przekazuje go kucharzowi:
class Order { // (obiekt transferu danych)
  private static int counter = 0;
  private final int id = counter++;
  private final Customer customer;
  private final WaitPerson waitPerson;
  private final Food food;
```

```
  public Order(Customer cust, WaitPerson wp, Food f) {
    customer = cust;
    waitPerson = wp;
    food = f;
  }
  public Food item() { return food; }
  public Customer getCustomer() { return customer; }
  public WaitPerson getWaitPerson() { return waitPerson; }
  public String toString() {
    return "Zamówienie: " + id + " na: " + food +
      " dla: " + customer +
      " obsługiwane przez: " + waitPerson;
  }
}

// A to otrzymujemy od kucharza:
class Plate {
  private final Order order;
  private final Food food;
  public Plate(Order ord, Food f) {
    order = ord;
    food = f;
  }
  public Order getOrder() { return order; }
  public Food getFood() { return food; }
  public String toString() { return food.toString(); }
}

class Customer implements Runnable {
  private static int counter = 0;
  private final int id = counter++;
  private final WaitPerson waitPerson;
  // Jedno danie naraz:
  private SynchronousQueue<Plate> placeSetting =
    new SynchronousQueue<Plate>();
  public Customer(WaitPerson w) { waitPerson = w; }
  public void
  deliver(Plate p) throws InterruptedException {
    // Wywołanie blokowane, jeśli klient wciąż
    // spożywa poprzednie danie:
    placeSetting.put(p);
  }
  public void run() {
    for(Course course : Course.values()) {
      Food food = course.randomSelection();
      try {
        waitPerson.placeOrder(this, food);
        // Blokowanie do momentu otrzymania dania:
        print(this + "spożywa " + placeSetting.take());
      } catch(InterruptedException e) {
        print("przerwano oczekiwanie " + this + " na " +
          course);
        break;
      }
    }
    print(this + "zakończył obiad, wychodzi");
  }
```

```
  public String toString() {
    return "Klient " + id + " ";
  }
}

class WaitPerson implements Runnable {
  private static int counter = 0;
  private final int id = counter++;
  private final Restaurant restaurant;
  BlockingQueue<Plate> filledOrders =
    new LinkedBlockingQueue<Plate>();
  public WaitPerson(Restaurant rest) { restaurant = rest; }
  public void placeOrder(Customer cust, Food food) {
    try {
      // Nie powinno blokować, bo to kolejka
      // LinkedBlockingQueue, bez ograniczenia pojemności:
      restaurant.orders.put(new Order(cust, this, food));
    } catch(InterruptedException e) {
      print(this + " przerwanie w metodzie placeOrder");
    }
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // Blokowanie do momentu gotowości dania
        Plate plate = filledOrders.take();
        print(this + "odebrał " + plate +
          " dla " +
          plate.getOrder().getCustomer());
        plate.getOrder().getCustomer().deliver(plate);
      }
    } catch(InterruptedException e) {
      print(this + " przerwany");
    }
    print(this + " po służbie");
  }
  public String toString() {
    return "Kelner " + id + " ";
  }
}

class Chef implements Runnable {
  private static int counter = 0;
  private final int id = counter++;
  private final Restaurant restaurant;
  private static Random rand = new Random(47);
  public Chef(Restaurant rest) { restaurant = rest; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // Blokowanie do momentu pojawienia się zamówienia:
        Order order = restaurant.orders.take();
        Food requestedItem = order.item();
        // Czas na przygotowanie zamówienia:
        TimeUnit.MILLISECONDS.sleep(rand.nextInt(500));
        Plate plate = new Plate(order, requestedItem);
        order.getWaitPerson().filledOrders.put(plate);
      }
```

```
      } catch(InterruptedException e) {
        print(this + " przerwany");
      }
      print(this + " po służbie");
    }
    public String toString() { return "Kucharz " + id + " "; }
  }

  class Restaurant implements Runnable {
    private List<WaitPerson> waitPersons =
      new ArrayList<WaitPerson>();
    private List<Chef> chefs = new ArrayList<Chef>();
    private ExecutorService exec;
    private static Random rand = new Random(47);
    BlockingQueue<Order>
      orders = new LinkedBlockingQueue<Order>();
    public Restaurant(ExecutorService e, int nWaitPersons,
      int nChefs) {
      exec = e;
      for(int i = 0; i < nWaitPersons; i++) {
        WaitPerson waitPerson = new WaitPerson(this);
        waitPersons.add(waitPerson);
        exec.execute(waitPerson);
      }
      for(int i = 0; i < nChefs; i++) {
        Chef chef = new Chef(this);
        chefs.add(chef);
        exec.execute(chef);
      }
    }
    public void run() {
      try {
        while(!Thread.interrupted()) {
          // Przybywa nowy klient; przypisać mu kelnera (WaitPerson):
          WaitPerson wp = waitPersons.get(
            rand.nextInt(waitPersons.size()));
          Customer c = new Customer(wp);
          exec.execute(c);
          TimeUnit.MILLISECONDS.sleep(100);
        }
      } catch(InterruptedException e) {
        print("Przerwano zadanie Restaurant");
      }
      print("Zamykanie restauracji");
    }
  }

  public class RestaurantWithQueues {
    public static void main(String[] args) throws Exception {
      ExecutorService exec = Executors.newCachedThreadPool();
      Restaurant restaurant = new Restaurant(exec, 5, 2);
      exec.execute(restaurant);
      if(args.length > 0) // Argument opcjonalny
        TimeUnit.SECONDS.sleep(new Integer(args[0]));
      else {
        print("Aby zakończyć, naciśnij 'Enter'");
        System.in.read();
```

```
    }
    exec.shutdownNow();
  }
} /* Output: (Sample)
Aby zakończyć, naciśnij 'Enter'
Kelner 0 odebrał SPRING_ROLLS dla Klient 1
Klient 1 spożywa SPRING_ROLLS
Kelner 3 odebrał SPRING_ROLLS dla Klient 0
Klient 0 spożywa SPRING_ROLLS
Kelner 0 odebrał BURRITO dla Klient 1
Klient 1 spożywa BURRITO
Kelner 3 odebrał SPRING_ROLLS dla Klient 2
Klient 2 spożywa SPRING_ROLLS
Kelner 3 odebrał BURRITO dla Klient 0
Klient 0 spożywa BURRITO
Kelner 1 odebrał SPRING_ROLLS dla Klient 3
Klient 3 spożywa SPRING_ROLLS
...
*///:~
```

Ważną rzeczą, jaką należałoby tu zaobserwować, jest zarządzanie złożonością poprzez zastosowanie kolejek do komunikacji pomiędzy zadaniami. Ta technika znakomicie upraszcza proces programowania współbieżnego przez odwrócenie kontroli: zadania nie wpływają na siebie nawzajem bezpośrednio — zadania wysyłają do siebie obiekty za pośrednictwem kolejek. Zadanie odbierające obsługuje (przetwarza) obiekt, traktując go jak komunikat, unikając „pobudzania" komunikatami. Wdrażając tę technikę, gdzie się da, można konstruować naprawdę solidne systemy współbieżne.

Ćwiczenie 36. Zmodyfikuj program *RestaurantWithQueues.java* tak, aby dla każdego stolika istniał jeden obiekt zamówienia zbiorczego — `OrderTicket`. Zmień `order` na `orderTicket` i dodaj klasę `Table` (stolik) z potencjalnie wieloma klientami przy stole (10).

Rozdzielanie zadań

Następny przykład symulacyjny będzie łączył wiele koncepcji omawianych w bieżącym rozdziale. Przedmiotem symulacji będzie hipotetyczna, zrobotyzowana linia montażowa w fabryce samochodów. Budowa każdego samochodu (obiektu `Car`) będzie odbywać się w kilku etapach: najpierw złożenie nadwozia, potem instalacja silnika, przeniesienia napędu i wreszcie kół.

```
//: concurrency/CarBuilder.java
// Skomplikowany przykład współdziałania zadań.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Car {
  private final int id;
  private boolean
    engine = false, driveTrain = false, wheels = false;
  public Car(int idn) { id = idn; }
  // Puste obiekty Car:
  public Car() { id = -1; }
  public synchronized int getId() { return id; }
  public synchronized void addEngine() { engine = true; }
```

```
  public synchronized void addDriveTrain() {
    driveTrain = true;
  }
  public synchronized void addWheels() { wheels = true; }
  public synchronized String toString() {
    return "Auto " + id + " [" + " silnik: " + engine
      + " napęd: " + driveTrain
      + " koła: " + wheels + " ]";
  }
}

class CarQueue extends LinkedBlockingQueue<Car> {}

class ChassisBuilder implements Runnable {
  private CarQueue carQueue;
  private int counter = 0;
  public ChassisBuilder(CarQueue cq) { carQueue = cq; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        TimeUnit.MILLISECONDS.sleep(500);
        // Składanie nadwozia:
        Car c = new Car(counter++);
        print("Zadanie ChassisBuilder zmontowało karoserię " + c);
        // Wstawianie do kolejki
        carQueue.put(c);
      }
    } catch(InterruptedException e) {
      print("Przerwanie: ChassisBuilder");
    }
    print("Zadanie ChassisBuilder wyłączone");
  }
}

class Assembler implements Runnable {
  private CarQueue chassisQueue, finishingQueue;
  private Car car;
  private CyclicBarrier barrier = new CyclicBarrier(4);
  private RobotPool robotPool;
  public Assembler(CarQueue cq, CarQueue fq, RobotPool rp){
    chassisQueue = cq;
    finishingQueue = fq;
    robotPool = rp;
  }
  public Car car() { return car; }
  public CyclicBarrier barrier() { return barrier; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        // Blokowanie do momentu zmontowania karoserii:
        car = chassisQueue.take();
        // Uruchomienie robotów:
        robotPool.hire(EngineRobot.class, this);
        robotPool.hire(DriveTrainRobot.class, this);
        robotPool.hire(WheelRobot.class, this);
        barrier.await(); // Oczekiwanie na zakończenie montażu
        // Umieszczenie gotowego auta w kolejce finishingQueue
        finishingQueue.put(car);
```

```
      }
    } catch(InterruptedException e) {
      print("Zadanie Assembler zakończone przerwaniem");
    } catch(BrokenBarrierException e) {
      // O tym wyjątku lepiej wiedzieć
      throw new RuntimeException(e);
    }
    print("Zadanie Assembler wyłączone");
  }
}

class Reporter implements Runnable {
  private CarQueue carQueue;
  public Reporter(CarQueue cq) { carQueue = cq; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        print(carQueue.take());
      }
    } catch(InterruptedException e) {
      print("Zadanie Reporter zakończone przerwaniem");
    }
    print("Zadanie Reporter wyłączone");
  }
}

abstract class Robot implements Runnable {
  private RobotPool pool;
  public Robot(RobotPool p) { pool = p; }
  protected Assembler assembler;
  public Robot assignAssembler(Assembler assembler) {
    this.assembler = assembler;
    return this;
  }
  private boolean engage = false;
  public synchronized void engage() {
    engage = true;
    notifyAll();
  }
  // Część metody run() różnicująca działanie robotów:
  abstract protected void performService();
  public void run() {
    try {
      powerDown(); // Oczekiwanie do następnego uruchomienia
      while(!Thread.interrupted()) {
        performService();
        assembler.barrier().await(); // Synchronizacja
        // Gotowe...
        powerDown();
      }
    } catch(InterruptedException e) {
      print("Zadanie " + this + " zakończone przerwaniem");
    } catch(BrokenBarrierException e) {
      // O tym wyjątku warto wiedzieć
      throw new RuntimeException(e);
    }
    print("Zadanie " + this + " wyłączone");
  }
```

```
  private synchronized void
  powerDown() throws InterruptedException {
    engage = false;
    assembler = null; // Odłączenie od linii montażowej
    // Umieszczamy się w kolejce robotów gotowych do pracy:
    pool.release(this);
    while(engage == false)  // Zasilanie wyłączone
      wait();
  }
  public String toString() { return getClass().getName(); }
}

class EngineRobot extends Robot {
  public EngineRobot(RobotPool pool) { super(pool); }
  protected void performService() {
    print(this + ": instalacja silnika");
    assembler.car().addEngine();
  }
}

class DriveTrainRobot extends Robot {
  public DriveTrainRobot(RobotPool pool) { super(pool); }
  protected void performService() {
    print(this + ": instalacja przeniesienia napędu");
    assembler.car().addDriveTrain();
  }
}

class WheelRobot extends Robot {
  public WheelRobot(RobotPool pool) { super(pool); }
  protected void performService() {
    print(this + ": instalacja kół");
    assembler.car().addWheels();
  }
}

class RobotPool {
  // Dyskretne zapobieganie dublowaniu wpisów:
  private Set<Robot> pool = new HashSet<Robot>();
  public synchronized void add(Robot r) {
    pool.add(r);
    notifyAll();
  }
  public synchronized void
  hire(Class<? extends Robot> robotType, Assembler d)
  throws InterruptedException {
    for(Robot r : pool)
      if(r.getClass().equals(robotType)) {
        pool.remove(r);
        r.assignAssembler(d);
        r.engage(); // Przekazanie do linii
        return;
      }
    wait(); // Brak dostępnych robotów
    hire(robotType, d); // Jeszcze jedna próba (rekurencyjna)
  }
```

```
  public synchronized void release(Robot r) { add(r); }
}
public class CarBuilder {
  public static void main(String[] args) throws Exception {
    CarQueue chassisQueue = new CarQueue(),
             finishingQueue = new CarQueue();
    ExecutorService exec = Executors.newCachedThreadPool();
    RobotPool robotPool = new RobotPool();
    exec.execute(new EngineRobot(robotPool));
    exec.execute(new DriveTrainRobot(robotPool));
    exec.execute(new WheelRobot(robotPool));
    exec.execute(new Assembler(
      chassisQueue, finishingQueue, robotPool));
    exec.execute(new Reporter(finishingQueue));
    // Uruchomienie całości przez montaż karoserii:
    exec.execute(new ChassisBuilder(chassisQueue));
    TimeUnit.SECONDS.sleep(7);
    exec.shutdownNow();
  }
} /* (Execute to see output) *///:~
```

Montowane samochody (obiekty `Car`) są transportowane pomiędzy miejscami montażu za pośrednictwem kolejki `CarQueue`, będącej podtypem `LinkedBlockingQueue`. Klasa `ChassisBuilder` zajmuje się montowaniem surowych aut (samych karoserii) umieszczanych w kolejce `CarQueue`. Zadanie `Assembler` odbiera auta z kolejki i zatrudnia roboty do montażu poszczególnych elementów. Zastosowana przy tym bariera `CyclicBarrier` pozwala na synchronizowanie odbioru pracy — `Assembler` czeka na barierze, aż wszystkie roboty zakończą montaż, po czym umieszcza gotowe auto w wyjściowej kolejce `CarQueue` transportującej samochody na następne stanowiska. Ostatecznym ich odbiorcą jest obiekt `Reporter`, który wypisuje informacje o odbieranych autach na wyjściu programu, co pozwala na sprawdzenie poprawności montażu.

Linia montażowa dysponuje pulą robotów; kiedy są potrzebne do montażu, zadanie `Assembler` angażuje odpowiedniego robota z puli. Po zakończeniu montażu robot trafia z powrotem do puli.

Metoda `main()` tworzy i inicjalizuje wszystkie niezbędne obiekty i zadania; ostatnim z nich jest `ChassisBuilder`, zadanie inicjujące proces montażu (ale z racji charakterystyki kolejki `LinkedBlockingQueue` równie dobrze zadanie to mogłoby być uruchamiane jako pierwsze). Zauważ, że program realizuje wszystkie podane w tym rozdziale wytyczne odnośnie zarządzania czasem życia obiektów i zadań, co umożliwia bezpieczne zatrzymanie całej linii montażowej.

Wszystkie metody klasy `Car` są metodami synchronizowanymi. Okazuje się, że w *tym konkretnym przykładzie* ta synchronizacja jest zbędna, bo w obrębie fabryki egzemplarze `Car` poruszają się w kolejce i w danym czasie nad danym obiektem `Car` zawsze pracuje tylko jedno zadanie. Zastosowanie odpowiednich kolejek wymusza szeregowanie dostępu do obiektów `Car`. Ale to typowy przykład pułapki: powiedzmy, że nie musimy synchronizować dostępu do obiektów klasy `Car`, bo i tak nie jest to tu potrzebne. Jednakże później, kiedy system zostanie połączony z innym, w którym synchronizacja `Car` będzie nieodzowna, całość się załamie.

Brian Goetz komentuje to tak:

> *Łatwiej powiedzieć: „Obiekt Car może być wykorzystywany w wielu wątkach, więc zabezpieczmy go na tę okoliczność w najbardziej oczywisty sposób". Osobiście porównuję to do barierek instalowanych wzdłuż ścieżek na krawędziach skarp na trasach widokowych; na niektórych z nich widnieją tabliczki: „Nie wychylać się". Oczywiście nie ma to chronić przed samym wychylaniem się, czy opieraniem o poręcz — chodzi o to, aby nie spaść ze zbocza. Ale przecież łatwiej wyrazić i egzekwować nakaz „nie wychylać się" niż być może bardziej odpowiedni nakaz: „nie spadać ze skarpy".*

Ćwiczenie 37. Zmodyfikuj program *CarBuilder.java* tak, aby zawierał dodatkowe stanowisko montażu, na którym samochód uzbrajano by w układ wydechowy, zderzaki i reflektory. Tak jak na drugim etapie montażu zadania te mogą być wykonywane współbieżnie przez niezależne roboty (2).

Ćwiczenie 38. Wzorując się na programie *CarBuilder.java*, oprogramuj model budowy domu (od robót przygotowawczych przez fundamenty po ściany itd.) (3).

Wydajność

W bibliotece `java.util.concurrent` z wydania Java SE5 znajduje się szereg klas mających na celu podnoszenie wydajności programów współbieżnych. Przeglądając zawartość biblioteki, trudno się zorientować, które z klas są przeznaczone do zwykłego użytku (jak klasy `BlockingQueue`), a które mają za jedyne zadanie zwiększać wydajność programu. Dlatego w tym podrozdziale zajmiemy się niektórymi aspektami strojenia aplikacji współbieżnej pod kątem wydajności i klasami, które takie strojenie umożliwiają.

Porównanie technologii muteksów

Skoro Java SE5 obok tradycyjnego słowa kluczowego `synchronized` zawiera również nowe klasy blokad `Lock` i `Atomic`, można pokusić się o porównanie podejścia klasycznego z nowym, poznając przy okazji zalety i dziedziny zastosowań obu rozwiązań.

Naiwne podejście polegałoby na prostym przetestowaniu każdego z modeli, jak tu:

```
//: concurrency/SimpleMicroBenchmark.java
// Ryzyko testowania w mikroskali.
import java.util.concurrent.locks.*;

abstract class Incrementable {
  protected long counter = 0;
  public abstract void increment();
}

class SynchronizingTest extends Incrementable {
  public synchronized void increment() { ++counter; }
}

class LockingTest extends Incrementable {
```

```
  private Lock lock = new ReentrantLock();
  public void increment() {
    lock.lock();
    try {
      ++counter;
    } finally {
      lock.unlock();
    }
  }
}

public class SimpleMicroBenchmark {
  static long test(Incrementable incr) {
    long start = System.nanoTime();
    for(long i = 0; i < 10000000L; i++)
      incr.increment();
    return System.nanoTime() - start;
  }
  public static void main(String[] args) {
    long synchTime = test(new SynchronizingTest());
    long lockTime = test(new LockingTest());
    System.out.printf("Słowo synchronized: %1$10d\n", synchTime);
    System.out.printf("Jawna blokada Lock: %1$10d\n", lockTime);
    System.out.printf("Lock/synchronized = %1$.3f",
      (double)lockTime/(double)synchTime);
  }
} /* Output: (75% match)
Słowo synchronized:  244919117
Jawna blokada Lock:  939098964
Lock/synchronized = 3.834
*///:~
```

Na wyjściu programu wywołania metod synchronizowanych zdają się szybsze od wywołań z użyciem blokady `ReentrantLock`. Skąd taki efekt?

Otóż ten przykład ilustruje ryzyko związane z tak zwanym „testowaniem w mikroskali”[23] (ang. *microbenchmarking*). Termin ten odnosi się do testowania pewnej cechy aplikacji w izolacji od pozostałych cech, niejako poza kontekstem właściwego użycia. Oczywiście nie oznacza to, że twierdzenia w rodzaju „blokady jawne są szybsze od metod synchronizowanych” nadal muszą być popierane właściwymi testami, trzeba jednak przy pisaniu tych testów wiedzieć, co faktycznie dzieje się i podczas kompilacji, i podczas wykonania.

Zaprezentowany przykład jest obarczony paroma błędami. Przede wszystkim prawdziwą różnicę wydajności pomiędzy różnymi implementacjami blokad zobaczymy dopiero wówczas, gdy blokady będą *w użyciu*, co oznacza obecność żądań, które będą próbowały podjąć wykonanie sekcji kodu chronionych muteksami. Tymczasem w powyższym przykładzie każdy z muteksów jest testowany w izolacji, w pojedynczym wątku metody `main()`.

[23] Wyjaśnił mi to Brian Goetz. Polecam jego artykuł publikowany pod adresem *http://www-128.ibm.com/developerworks/library/j-jtp12214*, w którym szerzej omawia zagadnienia pomiaru wydajności.

Po drugie, kompilator na widok słowa synchronized może wdrożyć specjalne optymalizacje, a nawet zorientować się, że program jest jednowątkowy. Kompilator może nawet wykryć, że wartość counter jest po prostu ileś razy zwiększana i zwyczajnie obliczy wynik tych inkrementacji, rezygnując z ich wykonywania. Różne kompilatory i różne systemy wykonawcze mogą się w tym aspekcie zasadniczo różnić, więc trudno dokładnie przewidzieć zachowanie programu; powinniśmy chociaż zapobiec możliwości wstępnego obliczenia wyniku zamiast jego wygenerowania.

Aby utworzyć poprawny test, powinniśmy bardziej skomplikować program. Przede wszystkim powinien on angażować większą liczbę zadań, i to nie tylko takich, które zmieniają wewnętrzne wartości, ale też i takich, które będą te wartości odczytywać (inaczej kompilator w ramach optymalizacji może się zorientować, że wartości nie są wcale używane). Do tego obliczenie musi być na tyle złożone i dawać na tyle nieprzewidywalny wynik, aby kompilator nie próbował agresywnej optymalizacji obliczeń. Osiągniemy to, ładując wstępnie obszerną tablicę losowych wartości typu int (wstępne ładowanie zmniejszy wpływ wywołań Random.nextInt() na wynik testów) i sumując wartości z tej tablicy:

```
//: concurrency/SynchronizationComparisons.java
// Porównanie wydajności jawnych blokad (Lock i Atomic)
// z wydajnością klasycznego mechanizmu synchronizacji metod.
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.concurrent.locks.*;
import java.util.*;
import static net.mindview.util.Print.*;

abstract class Accumulator {
  public static long cycles = 50000L;
  // Liczba zadań modyfikujących i odczytujących
  // uczestniczących w testach:
  private static final int N = 4;
  public static ExecutorService exec =
    Executors.newFixedThreadPool(N*2);
  private static CyclicBarrier barrier =
    new CyclicBarrier(N*2 + 1);
  protected volatile int index = 0;
  protected volatile long value = 0;
  protected long duration = 0;
  protected String id = "błąd";
  protected final static int SIZE = 100000;
  protected static int[] preLoaded = new int[SIZE];
  static {
    // Załadowanie tablicy wartości losowych:
    Random rand = new Random(47);
    for(int i = 0; i < SIZE; i++)
      preLoaded[i] = rand.nextInt();
  }
  public abstract void accumulate();
  public abstract long read();
  private class Modifier implements Runnable {
    public void run() {
      for(long i = 0; i < cycles; i++)
        accumulate();
      try {
        barrier.await();
```

```
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
    }
  }
  private class Reader implements Runnable {
    private volatile long value;
    public void run() {
      for(long i = 0; i < cycles; i++)
        value = read();
      try {
        barrier.await();
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
    }
  }
  public void timedTest() {
    long start = System.nanoTime();
    for(int i = 0; i < N; i++) {
      exec.execute(new Modifier());
      exec.execute(new Reader());
    }
    try {
      barrier.await();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
    duration = System.nanoTime() - start;
    printf("%-13s: %13d\n", id, duration);
  }
  public static void
  report(Accumulator acc1, Accumulator acc2) {
    printf("%-22s: %.2f\n", acc1.id + "/" + acc2.id,
      (double)acc1.duration/(double)acc2.duration);
  }
}

class BaseLine extends Accumulator {
  { id = "BaseLine"; }
  public void accumulate() {
    value += preLoaded[index++];
    if(index >= SIZE) index = 0;
  }
  public long read() { return value; }
}

class SynchronizedTest extends Accumulator {
  { id = "synchronized"; }
  public synchronized void accumulate() {
    value += preLoaded[index++];
    if(index >= SIZE) index = 0;
  }
  public synchronized long read() {
    return value;
  }
}
```

```
class LockTest extends Accumulator {
  { id = "Lock"; }
  private Lock lock = new ReentrantLock();
  public void accumulate() {
    lock.lock();
    try {
      value += preLoaded[index++];
      if(index >= SIZE) index = 0;
    } finally {
      lock.unlock();
    }
  }
  public long read() {
    lock.lock();
    try {
      return value;
    } finally {
      lock.unlock();
    }
  }
}

class AtomicTest extends Accumulator {
  { id = "Atomic"; }
  private AtomicInteger index = new AtomicInteger(0);
  private AtomicLong value = new AtomicLong(0);
  public void accumulate() {
    // Oho! Angażowanie więcej niż jednego egzemplarza Atomic
    // w danym czasie nie działa, ale daje pojęcie o wydajności:
    int i = index.getAndIncrement();
    value.getAndAdd(preLoaded[i]);
    if(++i >= SIZE)
      index.set(0);
  }
  public long read() { return value.get(); }
}

public class SynchronizationComparisons {
  static BaseLine baseLine = new BaseLine();
  static SynchronizedTest synch = new SynchronizedTest();
  static LockTest lock = new LockTest();
  static AtomicTest atomic = new AtomicTest();
  static void test() {
    print("==========================");
    printf("%-12s : %13d\n", "Liczba cykli", Accumulator.cycles);
    baseLine.timedTest();
    synch.timedTest();
    lock.timedTest();
    atomic.timedTest();
    Accumulator.report(synch, baseLine);
    Accumulator.report(lock, baseLine);
    Accumulator.report(atomic, baseLine);
    Accumulator.report(synch, lock);
    Accumulator.report(synch, atomic);
    Accumulator.report(lock, atomic);
  }
  public static void main(String[] args) {
    int iterations = 5; // Domyślnie
```

```
    if(args.length > 0) // Opcjonalna zmiana liczby iteracji
      iterations = new Integer(args[0]);
    // Wypełnienie puli wątków:
    print("Rozruch");
    baseLine.timedTest();
    // Teraz początkowy test nie ujmuje kosztu uruchomienia wątków.
    // Wygenerowanie poszczególnych punktów danych:
    for(int i = 0; i < iterations; i++) {
      test();
      Accumulator.cycles *= 2;
    }
    Accumulator.exec.shutdown();
  }
} /* Output: (Sample)
Rozruch
BaseLine :  34237033
============================
Liczba cykli :  50000
BaseLine :  20966632
synchronized :  24326555
Lock :  53669950
Atomic :  30552487
synchronized/BaseLine : 1.16
Lock/BaseLine : 2.56
Atomic/BaseLine : 1.46
synchronized/Lock : 0.45
synchronized/Atomic : 0.79
Lock/Atomic : 1.76
============================
Liczba cykli :  100000
BaseLine :  41512818
synchronized :  43843003
Lock :  87430386
Atomic :  51892350
synchronized/BaseLine : 1.06
Lock/BaseLine : 2.11
Atomic/BaseLine : 1.25
synchronized/Lock : 0.50
synchronized/Atomic : 0.84
Lock/Atomic : 1.68
============================
Liczba cykli :  200000
BaseLine :  80176670
synchronized :  5455046661
Lock :  177686829
Atomic :  101789194
synchronized/BaseLine : 68.04
Lock/BaseLine : 2.22
Atomic/BaseLine : 1.27
synchronized/Lock : 30.70
synchronized/Atomic : 53.59
Lock/Atomic : 1.75
============================
Liczba cykli :  400000
BaseLine :  160383513
synchronized :  780052493
Lock :  362187652
Atomic :  202030984
synchronized/BaseLine : 4.86
Lock/BaseLine : 2.26
```

```
Atomic/BaseLine     : 1.26
synchronized/Lock   : 2.15
synchronized/Atomic : 3.86
Lock/Atomic         : 1.79
============================
Liczba cykli :      800000
BaseLine    :   322064955
synchronized :  336155014
Lock        :   704615531
Atomic      :   393231542
synchronized/BaseLine : 1.04
Lock/BaseLine       : 2.19
Atomic/BaseLine     : 1.22
synchronized/Lock   : 0.47
synchronized/Atomic : 0.85
Lock/Atomic         : 1.79
============================
Liczba cykli :      1600000
BaseLine    :   650004120
synchronized :  52235762925
Lock        :   1419602771
Atomic      :   796950171
synchronized/BaseLine : 80.36
Lock/BaseLine       : 2.18
Atomic/BaseLine     : 1.23
synchronized/Lock   : 36.80
synchronized/Atomic : 65.54
Lock/Atomic         : 1.78
============================
Liczba cykli :      3200000
BaseLine    :   1285664519
synchronized :  96336767661
Lock        :   2846988654
Atomic      :   1590545726
synchronized/BaseLine : 74.93
Lock/BaseLine       : 2.21
Atomic/BaseLine     : 1.24
synchronized/Lock   : 33.84
synchronized/Atomic : 60.57
Lock/Atomic         : 1.79
*///:~
```

Program ten korzysta z wzorca projektowego *Template Method*[24] (metoda-wzorzec), co polega na ujęciu całości wspólnego kodu w klasie bazowej i wyizolowaniu kodu zmiennego do implementacji metod `accumulate()` i `read()` w klasach pochodnych. W każdej z klas pochodnych `SynchronizedTest`, `LockTest` i `AtomicTest` `accumulate()` i `read()` wyrażają inne sposoby implementacji wzajemnego wykluczania zadań.

Zadania programu są uruchamiane za pośrednictwem wykonawcy z ustaloną pulą wątków — `FixedThreadPool`; wykonawca podejmuje operację utworzenia wszystkich wątków na samym początku programu, tak aby nic zakłócał wyników pomiarów konstrukcją i inicjalizacją egzemplarzy `Thread`. Dla pewności pierwotny test jest powtarzany, a wyniki z pierwszego przebiegu są odrzucane.

[24] Zobacz *Thinking in Patterns* pod adresem *www.MindView.net*.

Do zgrania wszystkich zadań przed ogłoszeniem zakończenia testu służy bariera `CyclicBarrier`.

Słowo kluczowe `static` przy deklaracji tablicy służy do wstępnego ładowania tablicy liczb losowych, jeszcze przed rozpoczęciem testów. Dzięki temu eliminujemy z wyniku pomiarów również narzut generowania liczb pseudolosowych.

Każde wywołanie `accumulate()` powoduje przesunięcie do następnej pozycji tablicy `preLoaded` (z ewentualnym „zawinięciem" na jej początek) i dodanie kolejnej wartości losowej do zmiennej `value`. Obecność wielu zadań `Modifier` i `Reader` wymusza pożądany natłok odwołań do obiektu `Accumulator`.

Warto zaznaczyć, że w teście `AtomicTest` uznałem, że sytuacja jest zbyt skomplikowana, aby używać obiektów `Atomic` — zasadniczo, jeśli się okazuje, że trzeba zastosować więcej niż jeden egzemplarz `Atomic`, należałoby zazwyczaj darować sobie i wybrać jeden z bardziej konwencjonalnych muteksów (dokumentacja JDK stwierdza, że stosowanie obiektów `Atomic` jest uzasadnione tylko wtedy, kiedy krytyczne aktualizacje obiektu były strzeżone pojedynczą zmienną). Zdecydowałem jednak o pozostawieniu testu choćby po to, aby można było wyrobić sobie pogląd odnośnie wydajności klasy `Atomic`.

W metodzie `main()` test jest uruchamiany kilkukrotnie, przy czym użytkownik programu może zdecydować o zmianie domyślnej liczby powtórzeń (która wynosi 5). W każdym przebiegu liczba cykli testowych jest podwajana, dzięki czemu można oszacować zachowanie muteksów przy coraz dłuższych przebiegach programu. Wyprowadzane wyniki są raczej nieoczekiwane. W pierwszych czterech iteracjach słowo kluczowe `synchronized` wydaje się efektywniejsze niż blokady `Lock` i `Atomic`, ale po przekroczeniu wartości progowej blokowanie bazujące na słowie `synchronized` staje się wyraźnie nieefektywne, podczas gdy blokady `Lock` i `Atomic` zachowują mniej więcej proporcję do testu `BaseLine`, sprawując się znacznie lepiej niż słowo `synchronized`.

Trzeba pamiętać, że ten program ma jedynie wykazywać różnice pomiędzy różnymi implementacjami muteksów, a także o tym, że przedrukowane powyżej wyniki dotyczą jedynie konkretnego komputera i testu wykonanego w określonych okolicznościach. Gdyby trochę poeksperymentować, można by się przekonać o znacznych różnicach wyników przy zmianie liczby wątków i wydłużeniu testów. Niektóre optymalizacje czasu wykonania nie są bowiem podejmowane, póki program nie przepracuje pewnego czasu, który w przypadku programów serwerowych może się wydłużyć do kilku godzin.

To powiedziawszy, mogę z czystym sumieniem stwierdzić, że zazwyczaj blokada `Lock` jest wydajniejsza od wbudowanej synchronizacji na bazie słowa kluczowego `synchronized`; do tego narzut związany z synchronizacją wbudowaną może być różny, podczas gdy jawne obiekty blokad są w tym względzie dość spójne.

Czy oznacza to, że należy rezygnować ze stosowania słowa `synchronized`? Podejmując taką decyzję, trzeba uwzględnić dwa czynniki. Po pierwsze, w programie *SynchronizationComparisons.java* ciała metod blokowanych są stosunkowo zwięzłe. Generalnie jest to dobra praktyka — ochronę muteksami należy zawężać do jak najbardziej zwięzłych sekcji kodu. Ale w praktyce sekcje chronione mogą być rozleglejsze niż w prezentowanym przykładzie, więc udział czasu wykonania takich sekcji do łącznego czasu wykonania ciała metody będzie znacznie większy, co z kolei zmniejszy znaczenie narzutu

związanego z pozyskiwaniem i zwalnianiem muteksów i zupełnie zmieni wyniki. Najwyraźniej jedynym sposobem oszacowania korzyści i kosztów — ale dopiero, kiedy zajdzie potrzeba optymalizacji wydajności, nigdy wcześniej — jest wypróbowanie różnych rozwiązań.

Po drugie, lektura kodu prezentowanego w tym rozdziale uwidacznia, że stosowanie słowa kluczowego `synchronized` przyczynia się do istotnego polepszenia czytelności kodu w porównaniu ze schematem blokowanie-`try-finally`-zwolnienie blokady wymaganym przy stosowaniu jawnych blokad. Dlatego też tam, gdzie to zasadne, promowałem stosowanie słowa `synchronized`. W innym miejscu książki stwierdziłem, że kod jest czytany znacznie częściej niż pisany — przy programowaniu kod powinien służyć nie tyle do komunikacji z komputerem, co do komunikacji z innymi programistami, co czyni czytelność kodu istotną wartością. W efekcie najlepiej zaczynać od stosowania słowa `synchronized`, a jawne blokady wdrażać tylko tam, gdzie zachodzi wyraźna potrzeba przyspieszenia programu.

Na koniec: możliwość użycia klas `Atomic` w programie współbieżnym to spory bonus, ale trzeba pamiętać, że (co było widać w *SynchronizationComparisons.java*) obiekty klasy `Atomic` są przydatne jednie w najprostszych sytuacjach, to znaczy tam, gdzie występuje tylko jeden modyfikowany obiekt `Atomic`, niezależny od wszystkich pozostałych obiektów. I tutaj bezpieczniej jest zacząć od tradycyjnych muteksów, a obiekty `Atomic` wdrażać jedynie w ramach niezbędnej optymalizacji wydajności.

Kontenery bez blokad

W rozdziale „Kolekcje obiektów" zaznaczałem, że kontenery są podstawowymi narzędziami programistycznymi, a więc i narzędziami programowania współbieżnego. Dlatego też pierwsze kontenery Javy, jak `Vector` czy `Hashtable`, posiadały wiele metod synchronizowanych powodujących jednak niedopuszczalny narzut w aplikacjach jednowątkowych. W wydaniu Javy 1.2 nowa biblioteka kontenerów została zaimplementowana z pominięciem synchronizacji, a klasa `Collections` otrzymała rozmaite statyczne „synchronizowane" metody, do synchronizowania różnych typów kontenerów. Był to krok naprzód, bo dawał wybór odnośnie ochrony danego kontenera za pomocą synchronizacji; wciąż obserwowano narzut związany z blokowaniem za pomocą słowa kluczowego `synchronized`. Java SE5 wprowadziła kilka nowych klas kontenerów implementowanych pod kątem wydajności wielowątkowej z użyciem sprytnych technik eliminujących konieczność blokowania.

Pomysł na kontenery pozbawione blokad przedstawia się następująco: modyfikacje kontenerów mogą się odbywać równocześnie z odczytami, dopóki odczytujący postrzegają jedynie wyniki modyfikacji *zakończonych*. Sama modyfikacja przeprowadzana jest na kopii danych (a czasem kopii całego kontenera), niewidocznej dla odczytujących w czasie trwania zmian. Dopiero zakończenie i zatwierdzenie modyfikacji powoduje automatyczną wymianę nowych danych i włączenie jej do struktury głównej; od tego momentu są one widoczne dla czytających.

W kontenerze `CopyOnWriteArrayList` zapis elementu powoduje skopiowanie całej tablicy stanowiącej pamięć kontenera. Pierwotna tablica zostaje nietknięta i jest dostępna dla czytających w czasie modyfikowania kopii. Kiedy modyfikacja się zakończy, odbywa

się atomowa operacja wymiany tablic, po której czytający widzą już nową zawartość tablicy. Jedną z zalet kontenera `CopyOnWriteArrayList` jest to, że nie zgłasza on wyjątku `ConcurrentModificationException`, kiedy w użyciu jest wiele iteratorów współbieżnie przeglądających i modyfikujących listę. Eliminuje to znaną z przeszłości konieczność pisania specjalnego kodu na okoliczność takich wyjątków.

Kontener `CopyOnWriteArraySet` cechuje się podobnym zachowaniem, a to dzięki wewnętrznemu bazowaniu na klasie `CopyOnWriteArrayList`.

Podobne techniki wykorzystano też w klasach `ConcurrentHashMap` i `ConcurentLinkedQueue` — te kontenery również są dostępne do współbieżnego odczytu i zapisu, ale przy ewentualnej modyfikacji dochodzi do kopiowania jedynie fragmentów kontenera, a nie jego całości. Jednakże czytający wciąż nie postrzegają modyfikacji w toku, a jedynie dopiero po ich zakończeniu. Kontener `ConcurrentHashMap` nie zgłasza wyjątków `ConcurrentModificationException`.

Znów o wydajności

Dopóki kontener pozbawiony blokad będzie wykorzystywany głównie w operacjach odczytu, dopóty będzie znacznie szybszy od odpowiednika synchronizowanego właśnie z racji eliminowania narzutów związanych z pozyskiwaniem i eliminowaniem blokad. Dotyczy to również zastosowań z niewielkim udziałem operacji zapisu elementów takiego kontenera, choć tu doprecyzowanie pojęcia „niewielki udział" może być nieco problematyczne. W niniejszym punkcie przyjrzymy się różnicom wydajności omawianych kontenerów w różnych warunkach.

Zacznę od uogólnionego szkieletu testowania różnych typów kontenerów, w tym kontenerów z rodziny `Map`. Typ kontenera reprezentowany jest parametrem typowym `C`:

```
// : concurrency/Tester.java
// Szkielet testowania wydajności kontenerów w aplikacjach współbieżnych.
import java.util.concurrent.*;
import net.mindview.util.*;

public abstract class Tester<C> {
  static int testReps = 10;
  static int testCycles = 1000;
  static int containerSize = 1000;
  abstract C containerInitializer();
  abstract void startReadersAndWriters();
  C testContainer;
  String testId;
  int nReaders;
  int nWriters;
  volatile long readResult = 0;
  volatile long readTime = 0;
  volatile long writeTime = 0;
  CountDownLatch endLatch;
  static ExecutorService exec =
    Executors.newCachedThreadPool();
  Integer[] writeData;
  Tester(String testId, int nReaders, int nWriters) {
    this.testId = testId + " " +
```

```
        nReaders + "o " + nWriters + "z";
      this.nReaders = nReaders;
      this.nWriters = nWriters;
      writeData = Generated.array(Integer.class,
        new RandomGenerator.Integer(), containerSize);
      for(int i = 0; i < testReps; i++) {
        runTest();
        readTime = 0;
        writeTime = 0;
      }
    }
    void runTest() {
      endLatch = new CountDownLatch(nReaders + nWriters);
      testContainer = containerInitializer();
      startReadersAndWriters();
      try {
        endLatch.await();
      } catch(InterruptedException ex) {
        System.out.println("Przerwanie na barierze endLatch");
      }
      System.out.printf("%-27s %14d %14d\n",
        testId, readTime, writeTime);
      if(readTime != 0 && writeTime != 0)
        System.out.printf("%-27s %14d\n",
          "czas odczytu + czas zapisu =", readTime + writeTime);
    }
    abstract class TestTask implements Runnable {
      abstract void test();
      abstract void putResults();
      long duration;
      public void run() {
        long startTime = System.nanoTime();
        test();
        duration = System.nanoTime() - startTime;
        synchronized(Tester.this) {
          putResults();
        }
        endLatch.countDown();
      }
    }
    public static void initMain(String[] args) {
      if(args.length > 0)
        testReps = new Integer(args[0]);
      if(args.length > 1)
        testCycles = new Integer(args[1]);
      if(args.length > 2)
        containerSize = new Integer(args[2]);
      System.out.printf("%-27s %14s %14s\n",
        "Typ", "Czas odczytu", "Czas zapisu");
    }
  } ///:~
```

Abstrakcyjna metoda `containerInitializer()` zwraca zainicjalizowany kontener przeznaczony do testowania, którego referencja zachowywana jest w polu `testContainer`. Druga metoda abstrakcyjna, `startReadersAndWriters()`, uruchamia zadania odczytu i zapisu, które odczytują i modyfikują elementy kontenera w ramach testu. Aby uwi-

docznić efekt obciążenia blokady (w przypadku kontenerów synchronizowanych) oraz narzutu operacji zapisu (w kontenerach bezblokadowych), uruchamianych jest szereg testów różniących się liczbą odczytujących i zapisujących.

Konstruktor otrzymuje rozmaite informacje co do przebiegu testu (ich znaczenie, reprezentowane nazwami parametrów, nie powinno budzić wątpliwości), a potem kilkukrotnie (zgodnie z wartością `repetitions`) wywołuje metodę `runTest()`. Metoda `runTest()` tworzy barierę `CountDownLatch` (do zgrania zakończenia wszystkich testów w czasie), inicjalizuje kontener, a następnie wywołuje metodę `startReadersAndWriters()` i oczekuje na zakończenie.

Klasy „czytających" i „zapisujących" bazują na klasie `TestTask`, która mierzy czas wykonania abstrakcyjnej metody `test()`, a następnie w bloku synchronizowanym wywołuje metodę `putResults()` w celu zachowania wyników pomiaru.

Aby tak zmontowaną infrastrukturę wykorzystać (można w niej ponownie rozpoznać wzorzec projektowy *Template Method*), musimy wyprowadzić własną klasę dziedziczącą po klasie `Tester` dla konkretnego typu kontenera i udostępnić odpowiednie klasy `Reader` i `Writer`:

```
// : concurrency/ListComparisons.java
// {Args: 1 10 10} (szybka weryfikacja w czasie kompilacji)
// Szacunkowe porównanie wydajności wielowątkowej kontenerów List.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

abstract class ListTest extends Tester<List<Integer>> {
  ListTest(String testId, int nReaders, int nWriters) {
    super(testId, nReaders, nWriters);
  }
  class Reader extends TestTask {
    long result = 0;
    void test() {
      for(long i = 0; i < testCycles; i++)
        for(int index = 0; index < containerSize; index++)
          result += testContainer.get(index);
    }
    void putResults() {
      readResult += result;
      readTime += duration;
    }
  }
  class Writer extends TestTask {
    void test() {
      for(long i = 0; i < testCycles; i++)
        for(int index = 0; index < containerSize; index++)
          testContainer.set(index, writeData[index]);
    }
    void putResults() {
      writeTime += duration;
    }
  }
  void startReadersAndWriters() {
    for(int i = 0; i < nReaders; i++)
```

```
        exec.execute(new Reader());
      for(int i = 0; i < nWriters; i++)
        exec.execute(new Writer());
    }
  }

  class SynchronizedArrayListTest extends ListTest {
    List<Integer> containerInitializer() {
      return Collections.synchronizedList(
        new ArrayList<Integer>(
          new CountingIntegerList(containerSize)));
    }
    SynchronizedArrayListTest(int nReaders, int nWriters) {
      super("Synch. ArrayList", nReaders, nWriters);
    }
  }

  class CopyOnWriteArrayListTest extends ListTest {
    List<Integer> containerInitializer() {
      return new CopyOnWriteArrayList<Integer>(
        new CountingIntegerList(containerSize));
    }
    CopyOnWriteArrayListTest(int nReaders, int nWriters) {
      super("CopyOnWriteArrayList", nReaders, nWriters);
    }
  }

  public class ListComparisons {
    public static void main(String[] args) {
      Tester.initMain(args);
      new SynchronizedArrayListTest(10, 0);
      new SynchronizedArrayListTest(9, 1);
      new SynchronizedArrayListTest(5, 5);
      new CopyOnWriteArrayListTest(10, 0);
      new CopyOnWriteArrayListTest(9, 1);
      new CopyOnWriteArrayListTest(5, 5);
      Tester.exec.shutdown();
    }
  } /* Output: (Sample)
  Typ                    Czas odczytu   Czas zapisu
  Synch. ArrayList 10r 0w      232158294700        0
  Synch. ArrayList 9r 1w       198947618203   24918613399
  czas odczytu + czas zapisu =   223866231602
  Synch. ArrayList 5r 5w       117367305062   132176613508
  czas odczytu + czas zapisu =   249543918570
  CopyOnWriteArrayList 10r 0w      758386889         0
  CopyOnWriteArrayList 9r 1w       741305671    136145237
  czas odczytu + czas zapisu =     877450908
  CopyOnWriteArrayList 5r 5w       212763075   67967464300
  czas odczytu + czas zapisu =    68180227375
  *///:~
```

W teście `ListTest` klasy `Reader` i `Writer` realizują swoje zadania odnośnie kontenera `List<Integer>`. W metodzie `Reader.putResults()` zachowywana jest wartość `duration` i `result`, aby uniknąć niechcianej optymalizacji. Metoda `startReadersAndWriters()` definiowana jest tak, aby tworzyć i uruchamiać egzemplarze `Reader` i `Writer`.

Pochodna `ListTest` musi przesłaniać dziedziczoną metodę `containerInitializer()`, tak aby tworzyła i inicjalizowała stosowne kontenery testowe.

W metodzie `main()` można obserwować wariacje testów z różnymi liczbami czytających i zapisujących. Parametry testu można zmieniać za pośrednictwem argumentów wiersza wywołania programu przekazywanych do wywołania `Tester.initMain()`.

Domyślnie test składa się z dziesięciu przebiegów; w ten sposób można ustabilizować wyniki, które mogą się zmieniać w czasie choćby z powodu uruchamiania w maszynie wirtualnej Javy mechanizmu optymalizacji *hotspot*; trzeba też uwzględnić działanie odśmiecacza pamięci[25]. Przedrukowana powyżej próbka wyników została ograniczona do ostatniego przebiegu każdego testu. Na wyjściu widać, że synchronizowany kontener `ArrayList` cechuje się stabilnością wydajności niezależnie od liczby czytających i zapisujących — czytający konkurują o blokady z innymi czytającymi, tak jak czynią to zapisujący. W przypadku kontenera `CopyOnWriteArrayList` sytuacja wygląda zgoła inaczej: wydajność kontenera jest nieporównanie większa przy zerowej liczbie zapisujących, ale również wyraźnie większa od `ArrayList` nawet przy pięciu zapisujących. Zdaje się więc, że kontener `CopyOnWriteArrayList` można stosować z czystym sumieniem; narzut związany z kopiowaniem całości kontenera wydaje się mniejszy niż narzut wynikły z częstego synchronizowania kontenera. Oczywiście w konkretnej aplikacji wypadałoby przetestować zachowanie obu kontenerów — tylko to pozwoli ostatecznie upewnić się o słuszności decyzji.

Podkreślam, że przeprowadzony test nijak się ma do wielości różnych wdrożeń kontenerów w programach współbieżnych i w swoim systemie otrzymasz najprawdopodobniej znacząco odmienne wyniki. Powinieneś jednak wyrobić sobie choćby ogólny pogląd odnośnie względnej wydajności obu omawianych kontenerów.

Ponieważ kontener `CopyOnWriteArraySet` używa wewnętrznie kontenera `CopyOnWriteArrayList`, można domniemywać, że `CopyOnWriteArraySet` będzie się zachowywał podobnie — darujemy sobie osobny test.

Porównanie implementacji kontenerów asocjacyjnych

Gotową już infrastrukturę testową wykorzystamy tym razem do oszacowania wydajności synchronizowanej implementacji `HashMap` w porównaniu z wydajnością kontenera `ConcurrentHashMap`:

```
// : concurrency/MapComparisons.java
// {Args: 1 10 10} (szybka weryfikacja w czasie kompilacji)
// Oszacowanie wydajności kontenerów Map w kontekście współbieżności.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

abstract class MapTest
extends Tester<Map<Integer,Integer>> {
  MapTest(String testId, int nReaders, int nWriters) {
```

[25] Jako wprowadzenie do zagadnień testowania kodu z uwzględnieniem wpływu dynamicznych optymalizacji stosowanych w Javie polecam artykuł *www-128.ibm.com/developerworks/library/j-jtp12214*.

```
    super(testId, nReaders, nWriters);
  }
  class Reader extends TestTask {
    long result = 0;
    void test() {
      for(long i = 0; i < testCycles; i++)
        for(int index = 0; index < containerSize; index++)
          result += testContainer.get(index);
    }
    void putResults() {
      readResult += result;
      readTime += duration;
    }
  }
  class Writer extends TestTask {
    void test() {
      for(long i = 0; i < testCycles; i++)
        for(int index = 0; index < containerSize; index++)
          testContainer.put(index, writeData[index]);
    }
    void putResults() {
      writeTime += duration;
    }
  }
  void startReadersAndWriters() {
    for(int i = 0; i < nReaders; i++)
      exec.execute(new Reader());
    for(int i = 0; i < nWriters; i++)
      exec.execute(new Writer());
  }
}

class SynchronizedHashMapTest extends MapTest {
  Map<Integer,Integer> containerInitializer() {
    return Collections.synchronizedMap(
      new HashMap<Integer,Integer>(
        MapData.map(
          new CountingGenerator.Integer(),
          new CountingGenerator.Integer(),
          containerSize)));
  }
  SynchronizedHashMapTest(int nReaders, int nWriters) {
    super("Synch. HashMap", nReaders, nWriters);
  }
}

class ConcurrentHashMapTest extends MapTest {
  Map<Integer,Integer> containerInitializer() {
    return new ConcurrentHashMap<Integer,Integer>(
      MapData.map(
        new CountingGenerator.Integer(),
        new CountingGenerator.Integer(), containerSize));
  }
  ConcurrentHashMapTest(int nReaders, int nWriters) {
    super("ConcurrentHashMap", nReaders, nWriters);
  }
}
```

```
public class MapComparisons {
  public static void main(String[] args) {
    Tester.initMain(args);
    new SynchronizedHashMapTest(10, 0);
    new SynchronizedHashMapTest(9, 1);
    new SynchronizedHashMapTest(5, 5);
    new ConcurrentHashMapTest(10, 0);
    new ConcurrentHashMapTest(9, 1);
    new ConcurrentHashMapTest(5, 5);
    Tester.exec.shutdown();
  }
} /* Output: (Sample)
Typ                    Czas odczytu  Czas zapisu
Synch. HashMap 10r 0w      306052025049        0
Synch. HashMap 9r 1w       428319156207  47697347568
czas odczytu + czas zapisu =  476016503775
Synch. HashMap 5r 5w       243956877760  244012003202
czas odczytu + czas zapisu =  487968880962
ConcurrentHashMap 10r 0w    23352654318        0
ConcurrentHashMap 9r 1w     18833089400  1541853224
czas odczytu + czas zapisu =  20374942624
ConcurrentHashMap 5r 5w     12037625732  11850489099
czas odczytu + czas zapisu =  23888114831
*///:~
```

Wpływ dodawania zadań zapisujących jest w przypadku `ConcurrentHashMap` jeszcze bardziej znikomy niż w kontenerze `CopyOnWriteArrayList`; najwyraźniej w implementacji tego pierwszego wykorzystano inną technikę, jeszcze bardziej minimalizującą wpływ zapisów.

Blokowanie optymistyczne

Obiekty `Atomic` są przeznaczone do wykonywania operacji w rodzaju `decrementAndGet()`, ale niektóre klasy `Atomic` pozwalają też na przeprowadzanie czegoś, co zwie się *blokowaniem optymistycznym* (ang. *optimistic locking*). Oznacza to rezygnację ze stosowania blokady w czasie realizacji obliczeń i założenie muteksu dopiero wtedy, kiedy jesteśmy gotowi do aktualizacji obiektu `Atomic` za pośrednictwem metody `compareAndSet()`. W wywołaniu przekazuje się wartość poprzednią i wartość wyliczoną, a jeśli poprzednia nie zgadza się z bieżącą wartością obiektu `Atomic`, operacja okazuje się nieskuteczna — bo w międzyczasie jakieś inne zadanie zmodyfikowało obiekt i tym samym zdezaktualizowało obliczenia. Normalnie zapobiegalibyśmy równoczesnej modyfikacji obiektu przez różne zadania, ujmując całą operację ochroną muteksu (czy to w postaci blokady wbudowanej — `synchronized`, czy to jawnej — `Lock`), ale tu jesteśmy optymistami — zostawiamy dane niezablokowane w nadziei, że w potrzebnym nam krótkim czasie nie znajdzie się nikt chętny do ich modyfikowania. Wszystko to oczywiście w imię wydajności, bo użycie obiektu `Atomic` w miejsce operacji synchronizowanych (jawnie bądź niejawnie) powinno dać zysk wydajnościowy.

Co się dzieje, kiedy operacja `compareAndSet()` okaże się nieskuteczna? Tu sprawa się komplikuje i tu też tkwi ograniczenie optymistycznej techniki do problemów, które dadzą się ująć w pewnych ramach. Otóż jeśli wywołanie `compareAndSet()` zawiedzie, należy podjąć decyzję co do dalszego działania. To bardzo ważne: jeśli nie ma możliwości przywrócenia wcześniejszego stanu, należy zrezygnować z optymizmu i uciec się do stoso-

wania zwykłych muteksów. Być może właściwą reakcją będzie powtórzenie operacji i za drugim razem powiedzie się ona w całości. Być może odpowiednie będzie też najzwyklejsze zignorowanie niepowodzenia — w niektórych symulacjach utrata punktu danych i tak jest nadrabiana w późniejszych fazach (oczywiście taka decyzja wymaga szczegółowej znajomości modelowanego procesu).

Rozważmy fikcyjną symulację składającą się ze 100 000 „genów" o rozmiarze 30 elementów; powiedzmy, że to element jakiegoś algorytmu genetycznego. Załóżmy, że w każdej „ewolucji" algorytmu genetycznego odbywają się pewne bardzo kosztowne obliczenia, więc dla zwiększenia wydajności postanawiamy o rozprowadzeniu zadań pomiędzy procesorami systemu wieloprocesorowego. Dodatkowo zamiast blokad `Lock` zastosujemy obiekty `Atomic`, eliminując narzuty związane z obsługą jawnych muteksów (oczywiście wszystkie te decyzje zapadły dopiero po przetestowaniu pierwszej wersji kodu, zaimplementowanej w możliwie prosty sposób, kiedy to profilowanie wykazało niezbicie konieczność optymalizacji!). Z racji natury modelu w przypadku kolizji w czasie obliczeń zadanie, które wykryło kolizję, ignoruje ją i nie aktualizuje obliczonej wartości. Wygląda to tak:

```
//: concurrency/FastSimulation.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class FastSimulation {
  static final int N_ELEMENTS = 100000;
  static final int N_GENES = 30;
  static final int N_EVOLVERS = 50;
  static final AtomicInteger[][] GRID =
    new AtomicInteger[N_ELEMENTS][N_GENES];
  static Random rand = new Random(47);
  static class Evolver implements Runnable {
    public void run() {
      while(!Thread.interrupted()) {
        // Losowy wybór elementu do przetworzenia:
        int element = rand.nextInt(N_ELEMENTS);
        for(int i = 0; i < N_GENES; i++) {
          int previous = element - 1;
          if(previous < 0) previous = N_ELEMENTS - 1;
          int next = element + 1;
          if(next >= N_ELEMENTS) next = 0;
          int oldvalue = GRID[element][i].get();
          // Realizacja jakichś kosztownych obliczeń:
          int newvalue = oldvalue +
            GRID[previous][i].get() + GRID[next][i].get();
          newvalue /= 3; // Średnia trzech wartości
          if(!GRID[element][i]
            .compareAndSet(oldvalue, newvalue)) {
            // W przypadku niepowodzenia zakładamy jedynie jego
            // zgłoszenie i późniejsze zignorowanie; model obliczeń
            // prędzej czy później poradzi sobie z tym.
            print("Zmiana poprzedniej wartości (było: "
                  + oldvalue + ")");
          }
        }
```

```
      }
    }
  }
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < N_ELEMENTS; i++)
      for(int j = 0; j < N_GENES; j++)
        GRID[i][j] = new AtomicInteger(rand.nextInt(1000));
    for(int i = 0; i < N_EVOLVERS; i++)
      exec.execute(new Evolver());
    TimeUnit.SECONDS.sleep(5);
    exec.shutdownNow();
  }
} /* (Execute to see output) *///:~
```

Elementy są rozmieszczane wewnątrz tablicy z założeniem, że zwiększy to wydajność (założenie to poddamy weryfikacji w ramach ćwiczenia). Każdy obiekt `Evolver` uśrednia swoją wartość z obiektem poprzedzającym i następnym, a jeśli aktualizacja okaże się nieskuteczna, po prostu wypisuje wartość na wyjściu i przechodzi do uśredniania następnego elementu. Zauważ, że w programie nie pojawia się ani jeden muteks.

Ćwiczenie 39. Czy założenia, na których bazuje program *FastSimulation.java*, są poprawne? Spróbuj zmienić tablicę przetwarzanych elementów na tablicę najzwyklejszych wartości podstawowych `int` i użyć muteksów `Lock`. Porównaj wydajność obu wersji programu (6).

Blokady ReadWriteLock

Blokada `ReadWriteLock` optymalizuje sytuacje, w których struktura danych jest zapisywana stosunkowo rzadko, za to jest często odczytywana. Klasa `ReadWriteLock` pozwala na równoczesne zakładanie blokady odczytu przez wiele zadań czytających dopóty, dopóki nie pojawi się zadanie zapisujące. Po założeniu blokady zapisu żadne z zadań odczytujących nie uzyska dostępu aż do zwolnienia blokady zapisu.

Nie sposób powiedzieć, czy zastosowanie blokady `ReadWriteLock` faktycznie przyspieszy jakikolwiek program, bo zależy to od takich kwestii jak częstotliwość odczytu danych względem częstotliwości ich modyfikacji, stosunek czasu operacji odczytu do czasu operacji modyfikacji (blokada jest bardziej złożona, więc przy operacjach krótkotrwałych może nie ujawnić swoich zalet), ilość konkurujących zadań i dostępność wielu procesorów. Ostatecznie wpływ zastosowania klasy `ReadWriteLock` na wydajność programu należy po prostu oszacować eksperymentalnie.

Oto przykład ilustrujący podstawowy schemat zastosowania blokady `ReadWriteLock`:

```
//: concurrency/ReaderWriterList.java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ReaderWriterList<T> {
  private ArrayList<T> lockedList;
  // Dla większej sprawiedliwości przydziału blokady:
```

```
  private ReentrantReadWriteLock lock =
    new ReentrantReadWriteLock(true);
  public ReaderWriterList(int size, T initialValue) {
    lockedList = new ArrayList<T>(
      Collections.nCopies(size, initialValue));
  }
  public T set(int index, T element) {
    Lock wlock = lock.writeLock();
    wlock.lock();
    try {
      return lockedList.set(index, element);
    } finally {
      wlock.unlock();
    }
  }
  public T get(int index) {
    Lock rlock = lock.readLock();
    rlock.lock();
    try {
      // Pokazuje, że blokadę odczytu może
      // zakładać wielu czytających:
      if(lock.getReadLockCount() > 1)
        print(lock.getReadLockCount());
      return lockedList.get(index);
    } finally {
      rlock.unlock();
    }
  }
  public static void main(String[] args) throws Exception {
    new ReaderWriterListTest(30, 1);
  }
}

class ReaderWriterListTest {
  ExecutorService exec = Executors.newCachedThreadPool();
  private final static int SIZE = 100;
  private static Random rand = new Random(47);
  private ReaderWriterList<Integer> list =
    new ReaderWriterList<Integer>(SIZE, 0);
  private class Writer implements Runnable {
    public void run() {
      try {
        for(int i = 0; i < 20; i++) { // 2-sekundowy test
          list.set(i, rand.nextInt());
          TimeUnit.MILLISECONDS.sleep(100);
        }
      } catch(InterruptedException e) {
        // Dopuszczalny sposób przerwania
      }
      print("Zadanie zapisu zakończone");
      exec.shutdownNow();
    }
  }
  private class Reader implements Runnable {
    public void run() {
      try {
        while(!Thread.interrupted()) {
          for(int i = 0; i < SIZE; i++) {
```

```
            list.get(i);
            TimeUnit.MILLISECONDS.sleep(1);
          }
        }
      } catch(InterruptedException e) {
        // Dopuszczalny sposób przerwania
      }
    }
  }
  public ReaderWriterListTest(int readers, int writers) {
    for(int i = 0; i < readers; i++)
      exec.execute(new Reader());
    for(int i = 0; i < writers; i++)
      exec.execute(new Writer());
  }
} /* (Execute to see output) *///:~
```

Klasa `ReaderWriterList` może przechowywać listę elementów dowolnego typu, o stałym rozmiarze. Konstruktorowi trzeba przekazać pożądany rozmiar listy i obiekt początkowy, którym lista zostanie wypełniona. Metoda `set()` zakłada blokadę zapisu w celu oddelegowania wywołania do metody `ArrayList.set()`, a metoda `get()` zakłada blokadę odczytu, wywołując potem metodę `ArrayList.get()`. Dodatkowo metoda `get()` sprawdza, czy blokada odczytu została założona przez więcej niż jedno zadanie, a jeśli tak, wypisuje liczbę zadań, co ma uwidocznić możliwość równoczesnego odczytywania wartości blokowanej przez `ReadWriteLock`.

Aby przetestować klasę `ReaderWriterList`, klasa `ReaderWriterListTest` tworzy zadania odczytujące i zapisujące dla listy `ReaderWriterList<Integer>`. Zauważ, że zapisów jest w teście znacznie mniej niż odczytów.

Zaglądając do dokumentacji JDK dla klasy `ReentrantReadWriteLock`, przekonałbyś się, że klasa udostępnia sporo innych metod; dowiedziałbyś się też o zasadach „sprawiedliwości" blokowania. Okazuje się więc, że to narzędzie dość skomplikowane, a jako takie powinno być stosowane jedynie w poszukiwaniach sposobów zwiększenia wydajności działającego już programu. W pierwszym podejściu należałoby zaś stosować klasyczną synchronizację, uciekając się do blokad `ReadWriteLock` tylko w razie wyraźnej potrzeby.

Ćwiczenie 40. Naśladując przykład *ReaderWriterList.java*, utwórz klasę `ReaderWriterMap` bazującą na kontenerze `HashMap`. Przeanalizuj jego wydajność, wykorzystując odpowiednio zmodyfikowany program *MapComparisons.java*. Jak ma się synchronizowany kontener `HashMap` do `ConcurrentHashMap` (6)?

Obiekty aktywne

Przy lekturze bieżącego rozdziału masz zapewne wrażenie, że wielowątkowość w Javie to narzędzie skomplikowane i trudne w użyciu. Do tego zdaje się jakby nieproduktywne — choć zadania wykonują się współbieżnie, trzeba sporego wysiłku, aby je ze sobą zgrać i uniknąć kolizji oraz niepożądanych wzajemnych ingerencji.

Każdy, kto programował kiedyś w języku asemblerowym, przy programowaniu współbieżnym może mieć znajome wrażenie: liczy się każdy detal, za wszystko odpowiedzialny jest programista i nie ma nawet mowy o uprzęży zabezpieczającej przed upadkiem w postaci usług kontrolnych kompilatora.

Czyżby problemem był sam model wielowątkowości? W końcu jest zakorzeniony w świecie programowania proceduralnego. Być może istnieje inny model współbieżności, lepiej sprawdzający się w programowaniu obiektowym?

Jednym z modeli alternatywnych jest ten, który zakłada uczestnictwo *obiektów aktywnych* zwanych też *aktorami*[26]. Cecha „aktywności" oznacza tu samodzielne zarządzanie własnym wątkiem wykonania i własną kolejką komunikatów oraz założenie kolejkowania wszystkich żądań kierowanych do obicktu tak, aby były obsługiwane po kolei. Obiekty aktywne *szeregują komunikaty, a nie metody*, co oznacza, że można sobie darować ochronę przed problemami wynikającymi z przerwania wykonania zadania w środku jego pętli.

Przesłanie komunikatu do obiektu aktywnego powoduje przekształcenie tego komunikatu na zadanie, które trafia do kolejki obiektu w celu późniejszego uruchomienia. Do implementowania takiego modelu przyda się klasa Future wprowadzona Javie SE5. Oto prosty przykład z dwoma metodami kolejkującymi wywołania:

```
//: concurrency/ActiveObjectDemo.java
// Jako argumenty metod asynchronicznych można przekazywać
// stałe, wartości niezmienne, obiekty "rozłączone" albo
// inne obiekty aktywne.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ActiveObjectDemo {
  private ExecutorService ex =
    Executors.newSingleThreadExecutor();
  private Random rand = new Random(47);
  // Losowe opóźnienie dające efekt
  // prowadzenia długotrwałych obliczeń:
  private void pause(int factor) {
    try {
      TimeUnit.MILLISECONDS.sleep(
        100 + rand.nextInt(factor));
    } catch(InterruptedException e) {
      print("Przerwanie zadania w sleep()");
    }
  }
  public Future<Integer>
  calculateInt(final int x, final int y) {
    return ex.submit(new Callable<Integer>() {
      public Integer call() {
        print("zaczynam " + x + " + " + y);
        pause(500);
        return x + y;
      }
    });
```

[26] Dziękuję Allenowi Holubowi za niezbędne wyjaśnienia w tej materii.

```
  }
  public Future<Float>
  calculateFloat(final float x, final float y) {
    return ex.submit(new Callable<Float>() {
      public Float call() {
        print("zaczynam " + x + " + " + y);
        pause(2000);
        return x + y;
      }
    });
  }
  public void shutdown() { ex.shutdown(); }
  public static void main(String[] args) {
    ActiveObjectDemo d1 = new ActiveObjectDemo();
    // Blokada wyjątku ConcurrentModificationException:
    List<Future<?>> results =
      new CopyOnWriteArrayList<Future<?>>();
    for(float f = 0.0f; f < 1.0f; f += 0.2f)
      results.add(d1.calculateFloat(f, f));
    for(int i = 0; i < 5; i++)
      results.add(d1.calculateInt(i, i));
    print("Wykonano wszystkie wywołania asynchroniczne");
    while(results.size() > 0) {
      for(Future<?> f : results)
        if(f.isDone()) {
          try {
            print(f.get());
          } catch(Exception e) {
            throw new RuntimeException(e);
          }
          results.remove(f);
        }
    }
    d1.shutdown();
  }
} /* Output: (85% match)
Wykonano wszystkie wywołania asynchroniczne
zaczynam 0.0 + 0.0
zaczynam 0.2 + 0.2
0.0
zaczynam 0.4 + 0.4
0.4
zaczynam 0.6 + 0.6
0.8
zaczynam 0.8 + 0.8
1.2
zaczynam 0 + 0
1.6
zaczynam 1 + 1
0
zaczynam 2 + 2
2
zaczynam 3 + 3
4
zaczynam 4 + 4
6
8
*///:~
```

„Wykonawca pojedynczego wątku", generowany wywołaniem `Executors.newSingleThreadExecutor()`, zarządza własną, nieograniczoną co do liczby elementów kolejką i dysponuje tylko jednym wątkiem, w którym uruchamia zadania wyciągane kolejno z kolejki. W metodach `calculateInt()` i `calculateFloat()` musimy jedynie przekazać do wykonawcy (`submit()`) obiekt implementujący interfejs `Callable`, konwertując tym samym wywołania tych metod na komunikaty. Właściwe ciało każdej z tych metod jest ujęte w metodzie `call()` w anonimowej klasie wewnętrznej, definiującej ów obiekt `Callable`. Zauważ, że wartością zwracaną dla każdej metody obiektu aktywnego jest egzemplarz uogólnienia `Future` z parametrem typowym, którym jest właściwy typ zwracany metody. Dzięki temu omawiane metody mogą niemal natychmiast zwracać sterowanie do wywołującego, a wywołujący sprawdza moment zakończenia zadania i odwołuje się do wyników obliczenia za pośrednictwem otrzymanego obiektu `Future`. To najbardziej złożony przypadek takiej delegacji; dla metod niezwracających wartości proces jest jeszcze prostszy.

W metodzie `main()` tworzony jest kontener `List<Future<?>>` służący do przechowywania egzemplarzy `Future` pełniących rolę wyników komunikatów `calculateFloat()` i `calculateInt()`. Na rzecz elementów kontenera wykonywane są wywołania `isDone()`, a po zakończeniu zadań reprezentowanych przez poszczególne obiekty `Future` i przetworzeniu wyników obiekty te są usuwane z listy. Zwróć uwagę, że zastosowanie kontenera typu `CopyOnWriteArrayList` eliminuje konieczność jawnego kopiowania zawartości kontenera w celu uniknięcia wyjątków `ConcurrentModificationException`.

Aby dyskretnie zapobiec nadmiernym powiązaniom pomiędzy wątkami, wszelkie argumenty przekazywane do metod obiektu aktywnego powinny być wartościami niemodyfikowalnymi albo innymi obiektami aktywnymi, ewentualnie *obiektami rozłączonymi* (to mój własny termin), czyli takimi, które nie mają powiązania z żadnym innym zadaniem (trudno zapewnić ten wymóg, a to z powodu braku wsparcia ze strony języka).

Oto cechy obiektów aktywnych:

1. Każdy obiekt posiada własny wątek wykonania.
2. Każdy obiekt utrzymuje całkowitą kontrolę nad własnymi polami (to wymóg rygorystyczny względem zwykłych klas, które *mogą* chronić własne pola, ale nie muszą tego robić).
3. Całość komunikacji pomiędzy aktywnymi obiektami odbywa się w formie wymiany komunikatów pomiędzy tymi obiektami.
4. Komunikaty wymieniane pomiędzy obiektami aktywnymi są kolejkowane.

Rezultaty są całkiem przekonujące. Ponieważ operacja wysłania komunikatu od jednego obiektu aktywnego do innego takiego obiektu może być blokowana jedynie na czas wstawienia komunikatu do kolejki, a opóźnienie to jest zawsze bardzo krótkie i niezależne od wszelkich pozostałych obiektów, wysłanie komunikatu jest w praktyce operacją nieblokującą (w najgorszym przypadku blokującą na zaledwie moment). Ponieważ system obiektów aktywnych zakłada komunikację wyłącznie za pośrednictwem komunikatów, nie ma ryzyka zablokowania dwóch obiektów konkurujących o wywołanie metody na rzecz innego obiektu, a to oznacza brak możliwości wystąpienia zakleszczenia, co jest wielką zaletą. Ponieważ wątek wykonawczy w obrębie aktywnego obiektu obsługuje w danym czasie tylko jeden komunikat, nie dochodzi do rywalizacji o zasoby

i nie trzeba troszczyć się o synchronizowanie metod. Synchronizacja odbywa się i tak, ale na poziomie wymiany komunikatów, przez kolejkowanie wywołań metod i tym samym ich szeregowanie.

Niestety, przy braku bezpośredniego wsparcia ze strony kompilatora oprogramowanie opisanego schematu jest zbyt uciążliwe. Jednakże w dziedzinie obiektów aktywnych i aktorów obserwuje się ciągły postęp, podobnie jak w dziedzinie programowania agentowego. Agenty są obiektami aktywnymi, ale systemy agentowe obsługują również transparentną wymianę komunikatów pomiędzy maszynami połączonymi w sieć. Nie zdziwię się, jeśli programowanie agentowe wyprze kiedyś programowanie obiektowe, bo łączy obiektowość z względnie prostym rozwiązaniem problemów współbieżności.

O agentach, aktorach i obiektach aktywnych możesz przeczytać więcej w sieci WWW. Warte zainteresowania są zwłaszcza pomysły odnośnie obiektów aktywnych zawarte w teorii CSP (*Communicating Sequential Processes*) C.A.R. Hoare'a.

Ćwiczenie 41. Dodaj do programu *ActiveObjectDemo.java* obsługę metody (komunikatu) bez wartości zwracanej i wywołaj ją w metodzie `main()` (6).

Ćwiczenie 42. Zmodyfikuj program *WaxOMatic.java* tak, aby realizował koncepcję obiektów aktywnych (7).

Projekt[27]. Korzystając z adnotacji i Javassist, utwórz adnotację klasy `@Active` do transformacji klasy docelowej na obiekt aktywny.

Podsumowanie

Rozdział ten miał Cię wyposażyć w podstawową wiedzę o programowaniu współbieżnym za pomocą wątków Javy, abyś rozumiał, że:

1. Możesz uruchamiać wiele niezależnych zadań.
2. Musisz uwzględniać wszelkie możliwe problemy związane z zatrzymywaniem tych zadań
3. Zadania mogą wzajemnie wpływać na swoje działanie za pośrednictwem wspólnych zasobów. Podstawowym środkiem zapobiegawczym jest muteks (blokada).
4. Niestarannie zaprojektowane zadania mogą się wzajemnie zakleszczać.

Koniecznie należy dowiedzieć się, kiedy warto używać współbieżności, a kiedy jej unikać. Głównymi powodami przemawiającymi za stosowaniem współbieżności są:

- zarządzanie grupą wątków, których zastosowanie poprawi efektywność wykorzystania komputera (włącznie z możliwością transparentnego dla programisty rozprowadzania zadań pomiędzy wieloma dostępnymi procesorami),

[27] Proponowane projekty można wykorzystać na przykład jako warunek zaliczenia semestru. Rozwiązania dostępne dla zwykłych ćwiczeń nie zawierają oczywiście rozwiązań projektów.

- poprawienie organizacji kodu,
- poprawienie wygody obsługi programu.

Klasycznym przykładem równoważenia wykorzystania zasobów jest zrzekanie się procesora w czasie oczekiwania na zakończenie operacji wejścia-wyjścia. Klasycznym przykładem poprawiania wygody obsługi programu jest monitorowanie przycisku „stop" podczas długotrwałego pobierania plików.

Dodatkową zaletą wątków jest to, że udostępniają one „lekkie" zmiany kontekstu wykonywania (rzędu 100 instrukcji), a nie „ciężkie" zmiany kontekstu procesów (rzędu tysięcy instrukcji). Ponieważ wszystkie wątki w danym procesie współdzielą tę samą przestrzeń w pamięci, lekkie zmiany kontekstu modyfikują wyłącznie wykonywanie programu oraz zmienne lokalne wątków. Zmiany procesów — czyli ciężkie zmiany kontekstu — musiałyby także obejmować zmianę całości obszaru pamięci procesu.

Podstawowymi wadami wątków są:

1. Spowolnienie podczas oczekiwania wątków na zasoby współdzielone.
2. Dodatkowe wykorzystanie procesora związane z koniecznością obsługi wątków.
3. Nieproporcjonalny wzrost złożoności wynikający z błędnych decyzji projektowych.
4. Stwarzanie możliwości występowania patologii, takich jak zagłodzenia, sytuacje hazardowe czy zakleszczenie, tudzież tzw. *livelock* (czyli brak postępów zadania mimo formalnego braku zakleszczenia).
5. Występowanie niespójności pomiędzy różnymi platformami systemowymi. Na przykład, podczas tworzenia przykładów przedstawionych w niniejszej książce, wykryłem przypadki wyścigu, które bardzo szybko uwidaczniały się na jednych komputerach, lecz nie były zauważalne na innych. Gdybyś tworzył program na tym drugim komputerze, to po jego rozpowszechnieniu mógłbyś zostać bardzo niemile zaskoczony.

Jedna z największych trudności związanych z wątkami jest spowodowana tym, że więcej niż jedno zadanie może współdzielić jakiś zasób — jak na przykład pamięć w obiekcie — i musimy się upewnić, czy wiele zadań nie usiłuje czytać i zmieniać tego zasobu jednocześnie. Wymaga to rozważnego zastosowania dostępnych mechanizmów blokowania (w tym słowa kluczowego synchronized). To wielce przydatne narzędzia, ale trzeba je najpierw opanować, bo przy nieumiejętnym wykorzystaniu mogą dyskretnie doprowadzić do zakleszczenia zadań.

Ponadto z pewnością stworzenie aplikacji opartej na wątkach jest sztuką. Java została zaprojektowana, aby pozwalać na stworzenie tylu obiektów, ile tylko jest potrzebnych do rozwiązania zadania — przynajmniej teoretycznie (na przykład stworzenie milionów obiektów dla inżynierskiej analizy elementów skończonych może nie być możliwe do uzyskania w praktyce, jeśli nie uciekniemy się do wzorca projektowego *Flyweight*, czyli tzw. *wagi muszej*). Jednak wydaje się, że istnieje górna granica liczby wątków, które chcemy utworzyć, ponieważ z pewnych względów duża liczba wątków zdaje się nieporęczna. Ten punkt krytyczny jest trudny do określenia i często zależy od systemu operacyjnego i wirtualnej maszyny Javy; może on wypadać poniżej 100 bądź przekraczać

wiele tysięcy. Zazwyczaj, gdy tworzymy jedynie kilka wątków potrzebnych do rozwiązania zadania, nie stanowi to żadnego ograniczenia; niemniej jednak, w bardziej ogólnych projektach, może to być problemem wymuszającym wcielenie koncepcji współbieżności kooperacyjnej.

Niezależnie od tego, jak proste i atrakcyjne wydają się wątki w wydaniu niektórych języków programowania i bibliotek, powinieneś zawsze podchodzić do nich z szacunkiem, jak do czarnej magii. Zawsze spodziewaj się niespodziewanego. Problem ucztujących filozofów jest dlatego taki ciekawy, że można ustawić parametry problemu tak, aby zakleszczenie było niemal nieprawdopodobne, przez co rozwiązanie będzie miało fałszywe znamiona poprawności i niezawodności.

W ogólnym ujęciu wielowątkowość należy wykorzystywać wstrzemięźliwie i ostrożnie. Jeśli kwestie współbieżności stają się zbyt rozbudowane i skomplikowane, należałoby rozważyć wykorzystanie specjalnego języka, jak *Erlang*. To jeden z kilku dostępnych *języków funkcyjnych* specjalizowanych pod kątem wielowątkowości. Być może udałoby się zastosować taki język do implementacji tych części problemu, które wymagają podejścia wielowątkowego, jeśli rzecz nadmiernie się komplikuje.

Dalsza lektura

Niestety, w dziedzinie wielowątkowości funkcjonuje mnóstwo fałszywych poglądów — to tylko dowodzi stopnia złożoności czyhających tu problemów i łatwości nabycia fałszywego wrażenia ich ogarnięcia (wiem coś o tym, bo sam onegdaj uległem takiemu wrażeniu i nie mam wątpliwości, że zdarzy mi się to również w przyszłości). Sięgając po następną publikację o współbieżności, należy zawsze zachować pewną rezerwę oraz podejrzliwość i postarać się rozpoznać stopień rozeznania autora wśród omawianych kwestii. Jest oczywiście kilka takich książek, które mogę polecić z czystym sumieniem każdemu. Są to:

Java Concurrency in Practice, Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes i Doug Lea (Addison-Wesley, 2006). To doprawdy podstawowe źródło informacji o wielowątkowości w Javie.

Concurrent Programming in Java, Second Edition, Doug Lea (Addison-Wesley, 2000). Choć to książka grubo sprzed pojawienia się Javy SE5, to Doug walnie przyczynił się do implementacji biblioteki `java.util.concurrent`, więc ta publikacja to świetny materiał dla wszystkich chcących opanować kwestie współbieżności. Wykracza ona poza tematykę współbieżności w Javie i omawia współczesne nurty traktowania współbieżności w różnych językach programowania i technologiach. Miejscami ciężka, wymaga kilkukrotnej lektury (najlepiej w parotygodniowych odstępach czasu niezbędnych do przyswojenia i ułożenia materiału). Doug to jedna z niewielu osób na świecie, które faktycznie rozumieją współbieżność, i warto się od niego uczyć.

The Java Language Specification, Third Edition (rozdział 17.), Gosling, Joy, Steele i Bracha (Addison-Wesley, 2005). Specyfikacja techniczna, dostępna również w wygodnym wydaniu elektronicznym publikowanym pod adresem *http://java.sun.com/docs/books/jls*.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Rozdział 22.

Graficzne interfejsy użytkownika

Fundamentalna zasada projektowania brzmi: „Sprawiać, aby rzeczy proste były łatwe, a trudne były możliwe[1]*".*

Początkowo celem projektu bibliotcki graficznego interfejsu użytkownika (ang. *graphical user interface*, GUI) w Java 1.0 było umożliwienie programiście tworzenia interfejsu, który będzie wyglądał dobrze na wszystkich platformach. Tego celu nie udało się osiągnąć. Zamiast tego *Abstract Window Toolkit* (AWT) z Java 1.0 tworzy GUI, które wygląda równie źle we wszystkich systemach. Dodatkowo narzuca ograniczenia: można używać tylko czterech czcionek i nie można uzyskać dostępu do bardziej wyszukanych elementów GUI, które istnieją w danym systemie operacyjnym. Model programowania AWT Java 1.0 jest również niewygodny i niezorientowany obiektowo. Słuchacz jednego z moich seminariów (pracujący w Sunie w trakcie tworzenia Javy) wyjaśnił, dlaczego tak się stało — oryginalne AWT zostało wymyślone, zaprojektowane i zaimplementowane w ciągu miesiąca. Z pewnością jest to cud produktywności oraz praktyczna lekcja pokazująca, dlaczego ważne jest projektowanie.

Sytuacja poprawiła się wraz z wprowadzeniem modelu zdarzeń AWT Java 1.1, który charakteryzuje się o wiele sensowniejszym podejściem obiektowym, oraz standardu JavaBeans — komponentowego modelu programowania, ukierunkowanego na łatwe wykorzystanie w wizualnych środowiskach programistycznych. Java 2 (JDK 1.2) kończy transformację starego AWT Java 1.0 przez wymianę wszystkiego na *Java Foundation Classes* (JFC), którego część dotycząca GUI jest nazywana „Swing". Jest to bogaty zbiór łatwych w użyciu i zrozumiałych komponentów JavaBeans, z których metodą „przeciągnij i upuść" (jak również przez ręczne programowanie) można stworzyć (nareszcie) satysfakcjonujący interfejs. Wydaje się, że zasada „wersji trzeciej" przemysłu komputerowego (produkt nie jest dobry przed wersją trzecią) dotyczy również języków programowania.

---

[1] Inna wersja powyższego zdania, która mówi: „Nie rób użytkownikowi niespodzianek", nazwana jest „zasadą najmniejszego zaskoczenia".

Ten rozdział przedstawia zagadnienia dotyczące biblioteki Java Swing. Można bowiem założyć, że Swing, przynajmniej z punktu widzenia firmy Sun, jest ostateczną biblioteką GUI dla Javy[2]. Jeśli z jakiegoś powodu będziesz zmuszony używać „starego" AWT (ponieważ wymaga tego obsługa starszego kodu albo ze względu na ograniczenia przeglądarek), odpowiednie wprowadzenie możesz znaleźć w pierwszej wersji tej książki (do ściągnięcia pod adresem *www.MindView.net)*. Należy także pamiętać, że niektóre komponenty AWT wciąż są dostępne w Javie, a w niektórych sytuacjach korzystanie z nich jest konieczne.

Musisz wiedzieć, że nie jest to pełny przegląd komponentów Swing czy też metod opisywanych klas. To, co tu zaprezentuję, ma być prostym wprowadzeniem. Biblioteka Swing jest olbrzymia, a celem tego rozdziału jest jedynie wprowadzenie zagadnień niezbędnych i przybliżenie jej podstawowych pojęć. Jeśli trzeba zrobić coś więcej, to najprawdopodobniej okaże się, że Swing na to pozwala — o ile tylko będziesz miał ochotę poszukać rozwiązania.

Zakładam również, że pobrałeś i zainstalowałeś dokumentację bibliotek Javy w formacie HTML dostępną pod adresem *java.sun.com* i będziesz przeglądał opisy klas `javax.swing` w tej dokumentacji, aby zapoznać się ze szczegółami klas i metod biblioteki Swing. Możesz też przeszukiwać pod tym kątem sieć WWW, najlepiej jednak zacząć poszukiwania od przewodnika firmy Sun, publikowanego pod adresem *http://java.sun.com/docs/books/tutorial/uiswing*.

Dostępnych jest kilka (raczej grubych) książek poświeconych wyłącznie bibliotece Swing, po które możesz sięgnąć, gdy będziesz potrzebował bardziej wyczerpujących informacji bądź gdy konieczna będzie modyfikacja domyślnego sposobu działania biblioteki Swing.

W trakcie poznawania projektu Swing odkryjesz, że:

1. Swing jest prawdopodobnie o wiele lepszym modelem programowania niż te spotykane w innych językach i środowiskach programistycznych (nie jest doskonały, ale na pewno stanowi spory krok naprzód). JavaBeans (które zostaną przedstawione pod koniec tego rozdziału) stanowią „szkielet" tej biblioteki.
2. Narzędzia graficzne do tworzenia interfejsu użytkownika są *niezbędnym* elementem każdego środowiska programistycznego Javy. Biblioteki JavaBeans i Swing obsługują tego typu narzędzia, upraszczając automatyczne generowanie kodu, gdy użytkownik, używając graficznych narzędzi, umieszcza komponenty na formatkach. Powoduje to nie tylko znaczne przyspieszenie tworzenia interfejsu, ale umożliwia też przeprowadzanie wielu eksperymentów, a więc daje możliwość wypróbowania wielu projektów i przypuszczalnie znalezienia lepszego.
3. Prostota i dobry projekt biblioteki Swing gwarantują, że nawet używając narzędzi graficznych zamiast ręcznego kodowania, otrzymamy czytelny kod. Rozwiązuje to poważny problem związany z wcześniejszymi narzędziami graficznymi, które często tworzyły kod bardzo zagmatwany.

[2] Warto wiedzieć, że IBM stworzył na potrzeby swego środowiska Eclipse (*www.Eclipse.org*) nową, ogólnie dostępną bibliotekę graficzną, która może być stosowana jako alternatywa dla biblioteki Swing. Zajmiemy się nią w dalszej części rozdziału.

Swing zawiera wszelkie komponenty, które powinny się znajdować w nowoczesnym interfejsie użytkownika — od przycisków zawierających obrazki, po drzewa i tabele. Jest to duża biblioteka, ale tak zaprojektowana, aby jej złożoność była proporcjonalna do zastosowania — jeśli coś jest proste, to nie trzeba na to wiele kodu, jeśli próbuje się robić rzeczy bardziej skomplikowane, to kod staje się bardziej złożony.

W Swingu bardzo ważna jest „ortogonalność zastosowania". Znaczy to, że kiedy już pozna się ogólną koncepcję biblioteki, zazwyczaj można stosować ją wszędzie. Dzięki konwencjom nazewniczym, pisząc przykłady do bieżącego rozdziału, mogłem (w wielu przypadkach) odgadywać nazwy potrzebnych metod bez sięgania do dokumentacji. Jest to zdecydowanie efekt dobrego projektu biblioteki. W dodatku montowanie interfejsu sprowadza się często do podłączania jednych komponentów do drugich — a te później same działają tak jak trzeba.

Nawigacja za pomocą skrótów klawiaturowych jest automatyczna — można uruchomić aplikację Swing bez używania myszy i nie wymaga to żadnego dodatkowego programowania. Bez wysiłku można dodawać obsługę pasków przewijania (ang. *scrollbars*), opakowując komponent w panel `JScrollPane` podczas dodawania go do formatki. Użycie takich cech, jak podpowiedzi (ang. *tool tips*) wymaga najczęściej tylko jednego wiersza kodu.

Ze względów przenośności całość biblioteki Swing jest napisana w Javie.

Swing obsługuje również „wymienialne style interfejsu", co oznacza, że wygląd interfejsu użytkownika może być dynamicznie zmieniany, aby zaspokoić oczekiwania użytkowników pracujących na różnych platformach i systemach operacyjnych. Jest nawet możliwe (aczkolwiek trudne) stworzenie własnego stylu. Przykłady możesz znaleźć w sieci WWW[3].

Mimo wszystkich pozytywów Swing z pewnością nie jest dla każdego, nie rozwiązuje też wszelkich możliwych problemów projektowania interfejsów użytkownika. Pod koniec rozdziału przyjrzymy się więc dwóm alternatywom biblioteki Swing: sponsorowanej przez IBM (ale rozprowadzanej całkowicie nieodpłatnie) samodzielnej bibliotece SWT, wykorzystywanej w środowisku programistycznym Eclipse oraz narzędziu Flex firmy Macromedia, służącemu do przygotowywania flashowych interfejsów strony klienta dla aplikacji WWW.

Aplety

Kiedy Java ujrzała światło dzienne, najgłośniej było chyba o *apletach*, czyli programach, rozprowadzanych za pośrednictwem sieci Internet, a przeznaczonych do uruchamiania w przeglądarkach WWW (ze względów bezpieczeństwa środowisko uruchomieniowe apletu, tzw. piaskownica — ang. *sandbox* — jest mocno ograniczone). Ludzie widzieli w apletach Javy następny etap rozwoju internetu, a autorzy wielu książek poświęconych temu językowi zakładali wprost, że główną przyczyną zainteresowania programistów nowym językiem jest właśnie możliwość pisania apletów.

[3] Moim ulubionym jest styl „Napkin" (z ang. serwetka) Kena Arnolda. W tym stylu wszystkie elementy okien wyglądają, jakby były narysowane na serwetkach, z koślawymi czcionkami i znaczkami.

Z rozmaitych przyczyn wizje te nigdy nie stały się rzeczywistością. Jedną z przyczyn niepowodzenia rewolucji apletów był fakt nieobecności oprogramowania Javy niezbędnego do uruchamiania apletów na większości komputerów, zaś nie każdemu uśmiechało się ściąganie 10-megabajtowego pakietu po to, aby móc uruchomić coś, na co natrafił przypadkiem przy przeglądaniu stron WWW. Wielu użytkowników wystraszyło się samej koncepcji. Aplety Javy, jako mechanizm dystrybucji aplikacji strony klienta, nigdy nie przyjęły się masowo i choć tu i ówdzie wciąż widuje się je, stanowią już jedynie zaścianek technologii komputerowych.

Nie oznacza to, że aplety nie reprezentują ciekawej i cennej technologii. W sytuacji, w której istnieje pewność co do wyposażenia odbiorców oprogramowania w pakiet JRE (na przykład w środowisku korporacyjnym), aplety (albo komponenty JNLP tudzież Java Web Start, omawiane w dalszej części rozdziału) mogą stanowić świetny mechanizm rozprowadzania oprogramowania klienckiego i automatycznej jego aktualizacji bez kosztów i nakładów typowych dla ręcznej dystrybucji i instalacji nowego oprogramowania.

Wprowadzenie do technologii apletów znajdziesz w publikowanym w witrynie *www.MindView.net* suplemencie do niniejszej książki.

Podstawy biblioteki Swing

Większość aplikacji Swing będzie bazować na klasie `JFrame`, która tworzy okno aplikacji w danym systemie operacyjnym (będziemy je nazywać formatką). W wywołaniu konstruktora `JFrame` można określić zawartość paska tytułowego okna, jak tutaj:

```
//: gui/HelloSwing.java
import javax.swing.*;

public class HelloSwing {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Ahoj Swing");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 100);
    frame.setVisible(true);
  }
} ///:~
```

Metoda `setDefaultCloseOperation()` nakazuje egzemplarzowi `JFrame` wykonanie określonej operacji w reakcji na zdarzenie zamknięcia okna. Stała `EXIT_ON_CLOSE` oznacza zamknięcie programu. Bez wywołania tej metody aplikacja zastosowałaby operację domyślną, polegającą na braku reakcji — nie moglibyśmy zamknąć programu.

Metoda `setSize()` ustawia rozmiar okna (w pikselach).

Zwróć uwagę na ostatni wiersz:

```
frame.setVisible(true);
```

Bez niego na ekranie nie zobaczylibyśmy żadnego okna.

Aby program był trochę ciekawszy, możemy w oknie (JFrame) umieścić etykietę (JLabel):

```
//: gui/HelloLabel.java
import javax.swing.*;
import java.util.concurrent.*;

public class HelloLabel {
  public static void main(String[] args) throws Exception {
    JFrame frame = new JFrame("Ahoj Swing");
    JLabel label = new JLabel("Etykieta");
    frame.add(label);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 100);
    frame.setVisible(true);
    TimeUnit.SECONDS.sleep(1);
    label.setText("Hej! To coś innego!");
  }
} ///:~
```

Po upływie sekundy treść etykiety się zmienia. W tak prostych programach sterowanie komponentami interfejsu użytkownika z poziomu wątku głównego aplikacji (`main()`) jest zabawne i bezpieczne, ale zasadniczo nie jest to dobry pomysł. Biblioteka Swing uruchamia własny wątek przeznaczony do odbierania powiadomień o zdarzeniach interfejsu i aktualizacji wyglądu okna. Jeśli zaczniesz manipulować ekranem z poziomu innych wątków, możesz spowodować kolizje i zakleszczenia, o których pisałem w rozdziale „Współbieżność".

Owe inne wątki — w tym choćby wątek metody `main()` — powinny przekazywać zadania do wykonania przez *wątek dyspozytora zdarzeń* Swing[4]. Odbywa się to przez przekazanie zadania do metody `SwingUtilities.invokeLater()`, która umieszcza zadanie w *kolejce zdarzeń* (ang. *event queue*) do późniejszego uruchomienia przez wątek dyspozytora zdarzeń. Gdybyśmy zastosowali ten model do poprzedniego przykładu, otrzymalibyśmy coś takiego:

```
//: gui/SubmitLabelManipulationTask.java
import javax.swing.*;
import java.util.concurrent.*;

public class SubmitLabelManipulationTask {
  public static void main(String[] args) throws Exception {
    JFrame frame = new JFrame("Ahoj Swing");
    final JLabel label = new JLabel("Etykieta");
    frame.add(label);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 100);
    frame.setVisible(true);
    TimeUnit.SECONDS.sleep(1);
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        label.setText("Hej! To coś innego!");
      }
    });
  }
} ///:~
```

[4] Technicznie rzecz biorąc, dyspozytor ten pochodzi z biblioteki AWT.

Nie ma już mowy o bezpośrednim manipulowaniu etykietą. Zamiast tego w programie pojawił się obiekt implementujący interfejs Runnable, przekazany do obsłużenia do dyspozytora zdarzeń. W czasie realizacji zadania reprezentowanego obiektem dyspozytor nie wykonuje żadnych innych czynności, więc nie może dojść do żadnych kolizji — *pod warunkiem*, że całość kodu w programie trzyma się przyjętego modelu i sumiennie przekazuje zadania do dyspozytora za pośrednictwem metody SwingUtilities.invokeLater(). Dotyczy to również uruchamiania samego programu — metoda main() nie powinna wywoływać metod Swing, tak jak w powyższym programie, ale również przekazać zadanie do kolejki zdarzeń[5]. Prawidłowy program powinien więc wyglądać tak:

```
// : gui/SubmitSwingProgram.java
import javax.swing.*;
import java.util.concurrent.*;

public class SubmitSwingProgram extends JFrame {
  JLabel label;
  public SubmitSwingProgram() {
    super("Ahoj Swing");
    label = new JLabel("Etykieta");
    add(label);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(300, 100);
    setVisible(true);
  }
  static SubmitSwingProgram ssp;
  public static void main(String[] args) throws Exception {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() { ssp = new SubmitSwingProgram(); }
    });
    TimeUnit.SECONDS.sleep(1);
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        ssp.label.setText("Hej! To coś innego!");
      }
    });
  }
} ///:~
```

Zauważ, że wywołanie metody sleep() *nie* znajduje się w konstruktorze. Gdyby umieścić je w konstruktorze, etykieta w ogóle nie pojawiłaby się w oknie, bo konstruktor nie zakończyłby się przez zakończeniem czasu uśpienia. Umieszczenie wywołania sleep() w konstruktorze elementu interfejsu użytkownika albo w obrębie dowolnej operacji na tym interfejsie powoduje uśpienie wątku dyspozytora zdarzeń, co jest średnim pomysłem.

Ćwiczenie 1. Zmień program *HelloSwing.java* tak, aby sprawdzić, czy faktycznie bez wywołania setDefaultCloseOperation() okna programu nie można będzie zamknąć (1).

Ćwiczenie 2. Zmień program *HelloLabel.java* tak, aby pokazać, że etykieta jest dodawana do okna dynamicznie; spróbuj dodać do okna losowo dobraną liczbę etykiet (2).

[5] Praktyka ta pojawiła się wraz z wydaniem Javy SE5, więc w starszych programach można wciąż obserwować inne podejście. Nie oznacza to, że autorzy takich programów byli ignorantami. Zalecane praktyki programistyczne nie są zresztą stałe i ciągle ewoluują.

Platforma prezentacyjna

Łącząc poznane zalecenia możemy utworzyć platformę prezentacyjną dla wszystkich przykładów z biblioteki Swing; jej obecność zmniejszy ilość kodu w samych przykładach:

```
//: net/mindview/util/SwingConsole.java
// Narzędzie do uruchamiania programów przykładowych
// Swing (apletów i okien JFrame) z poziomu konsoli.
package net.mindview.util;
import javax.swing.*;

public class SwingConsole {
  public static void
  run(final JFrame f, final int width, final int height) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        f.setTitle(f.getClass().getSimpleName());
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(width, height);
        f.setVisible(true);
      }
    });
  }
} ///:~
```

Być może zechcesz skorzystać z tego narzędzia również we własnych projektach, dlatego zostało ono umieszczone w bibliotece `net.mindview.util`. Aby jej użyć, powinieneś osadzić swoją aplikację w oknie `JFrame` (jak wszystkie przykłady z niniejszej książki). Statyczna metoda `run()` ustawia pasek tytułowy okna zgodnie z nazwą klasy `JFrame`.

Ćwiczenie 3. Zmodyfikuj program *SubmitSwingProgram.java* tak, aby korzystał z platformy `SwingConsole` (3).

Tworzenie przycisku

Tworzenie przycisku jest całkiem proste: wystarczy wywołać konstruktor klasy `JButton` z etykietą, która ma być na przycisku. Później zobaczysz, jak można robić bardziej wymyślne rzeczy, jak choćby umieszczać obrazki na przyciskach.

Przeważnie tworzy się w klasie pole dla przycisku tak, by móc się później do niego odwoływać.

`JButton` jest komponentem — to tak naprawdę małe okno — który będzie automatycznie odrysowywany w trakcie odświeżania. Oznacza to, że nie trzeba wprost rysować przycisku lub jakiegokolwiek innego rodzaju kontrolki; wystarczy po prostu umieścić je na formatce i pozwolić im automatycznie zająć się rysowaniem samych siebie. Umieszczanie przycisku na formatce odbywa się zazwyczaj w jej konstruktorze:

```
//: gui/Button1.java
// Umieszczanie przycisków w oknach aplikacji Swing.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;
```

```
public class Button1 extends JFrame {
  private JButton
    b1 = new JButton("Przycisk 1"),
    b2 = new JButton("Przycisk 2");
  public Button1() {
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
  }
  public static void main(String[] args) {
    run(new Button1(), 200, 100);
  }
} ///:~
```

Pojawiło się tu coś nowego: przed umieszczeniem jakichkolwiek elementów na panelu zawartości nadaje mu się nowy „menedżer ułożenia komponentów" typu `FlowLayout`. Menedżer ułożenia dostarcza panelowi sposób na podejmowanie decyzji dotyczących umieszczenia kontrolki na formatce. Normalnym zachowaniem apletu jest użycie `BorderLayout`, ale nic zadziała to w tym przypadku, ponieważ (jak pokażę w dalszej części rozdziału) domyślnie całkowicie zakrywa każdą kontrolkę przez nowo dodaną. Natomiast `FlowLayout` powoduje, że kontrolki pływają swobodnie na formatce od lewej do prawej i od góry do dołu.

Ćwiczenie 4. Sprawdź, czy bez wywołania metody `setLayout()` w programie *Button1.java* w oknie programu widoczny byłby faktycznie tylko jeden przycisk (1).

Przechwytywanie zdarzenia

Po skompilowaniu i uruchomieniu powyższego apletu przy naciśnięciu przycisków nic się nie dzieje. Właśnie w tym miejscu należy wkroczyć i dopisać trochę kodu, aby określić, co powinno się stać. Podstawą programowania sterowanego zdarzeniami, obejmującego wiele zagadnień związanych z interfejsem użytkownika, jest wiązanie zdarzeń z kodem, który odpowiada na te zdarzenia.

W Swingu zostało to osiągnięte przez oddzielenie interfejsu (graficznych komponentów) od implementacji (kodu, który ma zostać wykonany, kiedy zajdzie zdarzenie dla komponentu). Każdy komponent Swinga może zgłaszać wszystkie zdarzenia, które mogą dla niego zachodzić, oraz zgłaszać poszczególne zdarzenia z osobna. Jeśli nie interesuje nas, na przykład, czy mysz jest przesuwana nad przyciskiem, nie wykazujemy zainteresowania tym zdarzeniem. Jest to bardzo prosta i elegancka metoda obsługi w programowaniu na podstawie zdarzeń i kiedy pozna się podstawy, można z łatwością używać komponentów, których w ogóle się wcześniej nie znało — w rzeczywistości model ten rozciąga się na wszystko, co może być sklasyfikowane jako JavaBean (które poznasz w dalszej części rozdziału).

Na początek skupmy się na najbardziej interesującym zdarzeniu używanego komponentu. W przypadku `JButton` tym „interesującym zdarzeniem" jest naciśnięcie przycisku. Aby nasze zainteresowanie wciśnięciem przycisku zostało zauważone (i zarejestrowane), wywołujemy metodę określającą odbiornik zdarzenia `addActionListener()` *klasy* `JButton`.

Metoda ta oczekuje argumentu, który jest obiektem implementującym interfejs `ActionListener`; interfejs ten składa się z pojedynczej metody o nazwie `actionPerformed()`. Zatem aby podłączyć kod do przycisku, wystarczy stworzyć klasę implementującą interfejs odbiornika `ActionListener` i zarejestrować obiekt tej klasy przez metodę `addActionListener()`. Metoda obsługi zostanie wywołana, kiedy tylko przycisk zostanie naciśnięty (normalnie określa się to mianem wywołania zwrotnego — ang. *callback*).

Tylko co ma być rezultatem naciśnięcia przycisku? Ponieważ chcielibyśmy zobaczyć, że coś się zmienia na ekranie, wprowadźmy nowy komponent Swing — `JTextField`. Jest to miejsce, w którym użytkownik może wpisać jakiś tekst lub — jak w tym przypadku — tekst może być zmodyfikowany przez program. Jest wiele sposobów stworzenia `JTextField`, ale w najprostszym wystarczy przekazać konstruktorowi szerokość pola. Po umieszczeniu pola `JTextField` na formatce można zmieniać jego zawartość, używając metody `setText()` (klasa `JTextField` ma jeszcze wiele innych metod, ich opis można znaleźć w dokumentacji JDK dostępnej pod adresem *java.sun.com*). Oto jak to wygląda:

```
//: gui/Button2.java
// Reagowanie na naciśnięcia przycisków.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Button2 extends JFrame {
  private JButton
    b1 = new JButton("Przycisk 1"),
    b2 = new JButton("Przycisk 2");
  private JTextField txt = new JTextField(10);
  class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String name = ((JButton)e.getSource()).getText();
      txt.setText(name);
    }
  }
  private ButtonListener bl = new ButtonListener();
  public Button2() {
    b1.addActionListener(bl);
    b2.addActionListener(bl);
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
    add(txt);
  }
  public static void main(String[] args) {
    run(new Button2(), 200, 150);
  }
} ///:~
```

Stworzenie pola tekstowego i umieszczenie go w obrębie okna wymaga podjęcia takich samych kroków, jak w przypadku przycisków `JButton` lub dowolnego komponentu Swing. Czymś nowym w powyższym programie jest stworzenie klasy `ButtonListener`, implementującej wspomniany wcześniej interfejs `ActionListener`. Argument metody `actionPerformed()` jest typu `ActionEvent` i zawiera wszelkie informacje o zdarzeniu oraz o tym, skąd ono nadeszło. Chciałem opisać, który przycisk został naciśnięty — metoda

getSource() zwraca obiekt, od którego pochodzi zdarzenie, i zakładam, że jest on typu JButton. Jego metoda getText() podaje tekst umieszczony na przycisku, który następnie zostaje umieszczony w polu tekstowym, aby pokazać, że kod rzeczywiście został wykonany jako reakcja na naciśnięcie przycisku.

Metoda addActionListener() wewnątrz konstruktora została użyta do zarejestrowania obiektu ButtonListener dla obydwu przycisków.

Przeważnie wygodniej jest zapisać ActionListener jako anonimową klasę wewnętrzną, zwłaszcza że staramy się używać tylko jednego egzemplarza każdego odbiornika. Przykład *Button2.java* można więc zmodyfikować tak, by wykorzystywał anonimowe klasy wewnętrzne:

```
//: gui/Button2b.java
// Zastosowanie anonimowych klas wewnętrznych.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Button2b extends JFrame {
  private JButton
    b1 = new JButton("Przycisk 1"),
    b2 = new JButton("Przycisk 2");
  private JTextField txt = new JTextField(10);
  private ActionListener bl = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      String name = ((JButton)e.getSource()).getText();
      txt.setText(name);
    }
  };
  public Button2b() {
    b1.addActionListener(bl);
    b2.addActionListener(bl);
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
    add(txt);
  }
  public static void main(String[] args) {
    run(new Button2b(), 200, 150);
  }
} ///:~
```

Sposób wykorzystujący anonimowe klasy wewnętrzne będzie używany (o ile to możliwe) w przykładach z tej książki.

Ćwiczenie 5. Utwórz aplikację korzystającą z klasy SwingConsole. Dołącz do niej jedno pole tekstowe i trzy przyciski. Naciśnięcia poszczególnych przycisków powinny zmieniać tekst wyświetlany w polu tekstowym (4).

Obszary tekstowe

Komponent JTextArea (obszar tekstowy) jest podobny do JTextField, tyle że może zawierać wiele wierszy tekstu i ma większe możliwości. Szczególnie przydatną metodą jest append(). Dzięki niej można łatwo wypisywać wyjście programu do JTextArea. Możliwość przewijania obszaru wstecz stanowi poprawę w stosunku do tego, co do tej pory osiągaliśmy, używając programów uruchamianych z wiersza poleceń, które wypisywały wyniki na standardowe wyjście. Jako przykład podaję poniższy program, który wypełnia pole JTextArea wynikami generatora geography wprowadzonego w rozdziale „Kontenery z bliska":

```
//: gui/TextArea.java
// Stosowanie kontrolki JTextArea.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

public class TextArea extends JFrame {
  private JButton
    b = new JButton("Dodaj dane"),
    c = new JButton("Wyczyść");
  private JTextArea t = new JTextArea(20, 40);
  private Map<String,String> m =
    new HashMap<String,String>();
  public TextArea() {
    // Użyj wszystkich danych:
    m.putAll(Countries.capitals());
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(Map.Entry me : m.entrySet())
          t.append(me.getKey() + ": "+ me.getValue()+"\n");
      }
    });
    c.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        t.setText("");
      }
    });
    setLayout(new FlowLayout());
    add(new JScrollPane(t));
    add(b);
    add(c);
  }
  public static void main(String[] args) {
    run(new TextArea(), 475, 425);
  }
} ///:~
```

W konstruktorze odwzorowanie Map jest wypełniane wszystkimi krajami i ich stolicami. Jak widać, dla obydwu przycisków tworzony jest odbiornik zdarzeń ActionListener, a następnie dodawany bez definiowania żadnej pośredniej zmiennej, ponieważ nie ma potrzeby

odwoływania się do niego ponownie w programie. Przycisk *Dodaj dane* formatuje i dopisuje wszystkie dane, podczas gdy przycisk *Wyczyść* wykorzystuje metodę `setText()`, aby usunąć cały tekst z obszaru `JTextArea`.

W trakcie dodawania `JTextArea` do `JFrame` jest on opakowywany przez panel o przesuwalnej zawartości `JScrollPane`, aby móc sterować przesuwaniem tekstu, gdy nie mieści się na ekranie. To wystarczy, aby zapewnić w pełni funkcjonujące przesuwanie. Po wielu próbach rozgryzienia, jak zrobić to samo w innych środowiskach programowania GUI, duże wrażenie robi na mnie prostota i dobry projekt komponentów takich jak `JScrollPane`.

Ćwiczenie 6. Zmodyfikuj program *strings/TestRegularExpression.java* w taki sposób, by stał się on w pełni interaktywnym programem wykorzystującym bibliotekę Swing, pozwalającym na zapisanie wejściowego łańcucha znaków w polu `TextArea` i wyrażenia regularnego w polu `TextField`. Uzyskane wyniki powinny być wyświetlane w drugim polu `TextArea` (7).

Ćwiczenie 7. Utwórz aplikację, używając klasy `SwingConsole` i umieść w niej wszystkie komponenty posiadające metodę `addActionListener()` (odszukaj je w dokumentacji HTML z *java.sun.com*. Podpowiedź: skorzystaj z indeksu). Przechwyć ich zdarzenia i dla każdego z nich wyświetl w polu tekstowym odpowiedni komunikat (5).

Ćwiczenie 8. Niemal każdy komponent Swing dziedziczy po klasie `Component` posiadającej metodę `setCursor()`. Odszukaj ją w dokumentacji JDK. Stwórz aplet i zmień kursor na jeden ze standardowych kursorów z klasy `Cursor` (6).

Rozmieszczanie elementów interfejsu

Sposób umieszczania komponentów na formatce stosowany w Javie różni się zapewne od wszystkich innych powszechnie stosowanych w systemach GUI. Po pierwsze, wszystko tutaj jest zapisane w kodzie; nie ma żadnych „zasobów" kontrolujących ułożenie komponentów. Po drugie, sposób ułożenia komponentów na formatce jest sterowany nie przez bezpośrednie pozycjonowanie, ale przez „menedżera ułożenia", który decyduje, jak komponenty mają być rozmieszczone, kierując się kolejnością ich dodawania (metodą `add()`). Rozmiar, kształt i ułożenie komponentów będą się znacznie różnić, zależnie od menedżera ułożenia. Dodatkowo menedżery ułożenia dostosowują wymiary apletu do okna aplikacji, więc jeśli rozmiar okna zostanie zmieniony, to rozmiar, kształt i położenie komponentów również może się zmienić.

Komponenty `JApplet`, `JFrame`, `JWindow` i `JDialog` mogą zawierać i wyświetlać komponenty (obiekty klas odziedziczonych z `Component`). W klasie `Container` dostępna jest metoda o nazwie `setLayout()`, która pozwala na wybranie jednego z menedżerów ułożenia. W tej części postaram się przybliżyć specyfikę różnych menedżerów ułożenia, zamieszczając w nich kilka przycisków (bo to chyba najprostsza czynność). Nie będzie tu przechwytywania zdarzeń przycisków, ponieważ te przykłady mają jedynie prezentować sposób układania elementów interfejsu.

BorderLayout

Wszystkie formatki JFrame używają tego układu jako domyślnego. Użyty bez żadnych dodatkowych instrukcji powoduje, że wszystko, co zostanie dodane przez metodę add(), będzie umieszczone na środku obszaru i rozciągnięte w każdym kierunku aż do krawędzi formatki.

Ten menedżer ułożenia kieruje się pojęciem czterech rejonów brzegowych oraz obszaru środkowego. Dodając coś do panelu używającego BorderLayout, można skorzystać z przeciążonej wersji metody add(), która przyjmuje jako pierwszy argument stałą. Może ona przyjąć jedną z poniższych wartości:

Stała	Znaczenie
BorderLayout.NORTH	góra
BorderLayout.SOUTH	dół
BorderLayout.EAST	prawo
BorderLayout.WEST	lewo
BorderLayout.CENTER	wypełnij środek, rozciągając do innych komponentów lub do krawędzi

Jeśli nie zostanie podany obszar, na którym ma zostać umieszczony obiekt, to domyślnie zostanie użyte CENTER.

Oto prosty przykład, w którym używany jest układ domyślny, ponieważ JFrame domyślnie używa właśnie BorderLayout:

```
//: gui/BorderLayout1.java
// Efekt działania menedżera układu BorderLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class BorderLayout1 extends JFrame {
  public BorderLayout1() {
    add(BorderLayout.NORTH, new JButton("Góra"));
    add(BorderLayout.SOUTH, new JButton("Dół"));
    add(BorderLayout.EAST, new JButton("Prawo"));
    add(BorderLayout.WEST, new JButton("Lewo"));
    add(BorderLayout.CENTER, new JButton("Środek"));
  }
  public static void main(String[] args) {
    run(new BorderLayout1(), 300, 250);
  }
} ///:~
```

Dla każdego ułożenia oprócz CENTER dodany element zostaje ścieśniony tak, by zmieścił się na jak najmniejszym obszarze wzdłuż jednego kierunku, podczas gdy jest maksymalnie rozciągany wzdłuż drugiej osi. Natomiast przy CENTER element jest rozciągany w obydwu kierunkach tak, by zajął środek.

FlowLayout

Ten menedżer po prostu „wypełnia" formatkę komponentami od lewej do prawej, dopóki nie zostanie zapełniona przestrzeń na górze, a później przechodzi do następnego wiersza i kontynuuje zapełnianie.

W poniższym przykładzie wybierany jest menedżer `FlowLayout`, a następnie na formatce umieszczane są cztery przyciski. Widać, że przy `FlowLayout` komponenty przyjmują swój „naturalny" rozmiar. Na przykład przycisk `JButton` będzie miał rozmiar umieszczonego na nim napisu.

```
//: gui/FlowLayout1.java
// Efekt działania menedżera układu FlowLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class FlowLayout1 extends JFrame {
  public FlowLayout1() {
    setLayout(new FlowLayout());
    for(int i = 0; i < 20; i++)
      add(new JButton("Przycisk " + i));
  }
  public static void main(String[] args) {
    run(new FlowLayout1(), 300, 300);
  }
} ///:~
```

Przy układzie `FlowLayout` wszystkie komponenty zostaną ściągnięte do ich najmniejszego rozmiaru, więc czasem można uzyskać trochę zaskakujące zachowanie. Na przykład, ponieważ etykieta `JLabel` będzie miała rozmiar umieszczonego na niej tekstu, próba wyrównania tego tekstu do prawej nie spowoduje zmiany na ekranie.

Zauważ, że zmiana rozmiaru okna spowoduje automatyczne dopasowanie i ewentualne przemieszczenie elementów przez menedżer układu.

GridLayout

Menedżer `GridLayout` pozwala utworzyć tabelę komponentów. W trakcie dodawania komponenty są umieszczane w tabeli od lewej do prawej i od góry do dołu. W konstruktorze można podać potrzebną liczbę wierszy i kolumn, które będą rozmieszczone równomiernie na obszarze panelu.

```
//: gui/GridLayout1.java
// Efekt działania menedżera układu GridLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class GridLayout1 extends JFrame {
  public GridLayout1() {
    setLayout(new GridLayout(7,3));
    for(int i = 0; i < 20; i++)
      add(new JButton("Przycisk " + i));
  }
```

```
  public static void main(String[] args) {
    run(new GridLayout1(), 300, 300);
  }
} ///:~
```

W tym przypadku mamy 21 gniazd, ale tylko 20 przycisków. Ostatnie gniazdo pozostaje puste, ponieważ `GridLayout` nie wykonuje „równoważenia".

GridBagLayout

Menedżer `GridBagLayout` dostarcza ogromnych możliwości kontroli tego, jak zostaną ułożone obszary okna oraz jak się mają poprzekształcać, kiedy okno zmieni rozmiar. Jednak jest to też najbardziej skomplikowany menedżer układu, raczej trudny do zrozumienia. Przeznaczony jest głównie do automatycznego generowania kodu przez graficzne narzędzia tworzenia interfejsu (dobre programy będą używały `GridBagLayout` zamiast pozycjonowania bezpośredniego). Jeśli tworzony projekt jest tak skomplikowany, że zachodzi potrzeba użycia `GridBagLayout`, to do jego wygenerowania lepiej wykorzystać takie narzędzia. Jeśli mimo to chciałbyś poznać zawiłe szczegóły tego menedżera, odsyłam do jednej z książek poświęconych w całości bibliotece Swing.

Jako alternatywę polecałbym rozważenie menedżera `TableLayout`, który *nie* wchodzi w skład biblioteki Swing, ale który można pobrać ze strony *http://java.sun.com*. Komponent ten bazuje na `GridBagLayout`, ukrywając większość jego złożoności i znacznie upraszczając stosowanie.

Pozycjonowanie bezpośrednie

Możliwe jest również bezpośrednie ustawienie pozycji komponentów graficznych. Można to zrobić w następujący sposób:

1. Usuń menedżera układu dla danego obiektu `Container`: `setLayout(null)`.
2. Dla każdego komponentu wywołaj `setBounds()` lub `reshape()` (zależnie od wersji języka), przekazując prostokąt ograniczający opisany w pikselach. Można to zrobić w konstruktorze lub metodzie `paint()`, w zależności od tego, co chce się uzyskać.

Niektóre programy do tworzenia GUI intensywnie wykorzystują to rozwiązanie, ale przeważnie nie jest to najlepszy sposób generowania kodu.

BoxLayout

Wiele osób miało poważne problemy ze zrozumieniem i używaniem menedżera `GridBagLayout`, dlatego Swing zawiera również menedżer `BoxLayout` posiadający wiele zalet `GridBagLayout`, *ale* nie tak skomplikowany. Można go więc częściej stosować, gdy trzeba ręcznie zakodować układ (jeśli projekt staje się zbyt skomplikowany, lepiej skorzystać z programu automatycznie generującego układ `GridBagLayout`). Menedżer `BoxLayout` pozwala na kontrolę rozmieszczenia komponentów w pionie lub w poziomie oraz na kontrolę odstępów pomiędzy komponentami przez stosowanie „rozdzielaczy i kleju". Przykłady można obejrzeć w suplementach do książki publikowanych na stronach *www.MindView.net*.

Najlepsze rozwiązanie?

Swing jest bardzo użytecznym narzędziem: kilka wierszy kodu może bardzo wiele zrobić. Przykłady przedstawione w tej książce są uproszczone i dlatego sensowne jest — w celu nauki — pisanie ich ręcznie. W rzeczywistości można wiele osiągnąć, łącząc proste układy. Jednak od pewnego momentu ręczne kodowanie formatek interfejsu użytkownika przestaje być sensowne — staje się zbyt skomplikowane i nie jest najlepszym spożytkowaniem czasu poświęconego na programowanie. Projektanci Javy i Swinga ukierunkowali język i biblioteki tak, by wspomagały narzędzia do graficznego tworzenia interfejsu użytkownika, które mają sprawić, że programowanie będzie łatwiejsze. Jeśli programista rozumie, co się dzieje z menedżerem układu, i wie, jak obsłużyć zdarzenia (opisane dalej), to nie jest szczególnie ważne czy wie, jak ułożyć komponenty ręcznie — niech robi to za nas odpowiednie narzędzie (przecież Java została zaprojektowana, by zwiększać wydajność programisty).

Model zdarzeń w Swingu

W modelu zdarzeń biblioteki Swing komponent może zainicjować zdarzenie. Każdy typ zdarzenia jest reprezentowany przez oddzielną klasę. Kiedy zdarzenie zostanie uruchomione, otrzymuje je jeden lub więcej „odbiorników", które reagują na nie. Zatem źródło zdarzenia i miejsce, w którym zdarzenie jest obsługiwane, mogą być rozdzielone. Przeważnie używa się komponentów Swing w takiej postaci, w jakiej są, i trzeba jedynie dopisać kod, który jest wywoływany, kiedy komponenty otrzymują zdarzenie. Dlatego jest to doskonały przykład oddzielenia interfejsu od implementacji.

Każdy odbiornik zdarzeń jest obiektem klasy implementującej szczególnego typu interfejs odbiornika. Zatem wszystko, co musi zrobić programista, to stworzyć obiekt odbiornika i zarejestrować go w komponencie, który wysyła zdarzenie. Rejestrację wykonuje się, wywołując metodę `addXXXListener()` komponentu wysyłającego zdarzenie, gdzie `XXX` reprezentuje typ nasłuchiwanego zdarzenia. Przeglądając nazwy metod typu `addListener`, można się łatwo dowiedzieć, jakiego rodzaju zdarzeń można nasłuchiwać, a próba nasłuchiwania niewłaściwych zakończy się błędem kompilacji. Później okaże się, że również komponenty JavaBean używają nazw metod `addListener`, aby określić, jakie zdarzenia może obsłużyć dany komponent.

Cała logika obsługi zdarzeń znajduje się zatem w klasie odbiornika. Jedynym ograniczeniem przy tworzeniu klasy odbiornika jest to, że musi ona implementować odpowiedni interfejs. Można tworzyć globalne klasy odbiorników, ale w tej sytuacji przydatne stają się klasy wewnętrzne. I to nie tylko dlatego, że pozwalają na logiczne pogrupowanie klas odbiorników wewnątrz obsługiwanych przez nie klas interfejsu użytkownika lub klas logiki biznesowej, ale również dlatego, że obiekty klasy zagnieżdżonej przechowują referencję do swoich obiektów nadrzędnych, a to umożliwia stosowanie odwołań do obiektu klasy zewnętrznej, z pominięciem granic istniejących pomiędzy klasami.

Wszystkie przedstawione dotychczas w tym rozdziale przykłady stosowały model zdarzeń Swing, a pozostała część rozdziału uzupełni szczegóły tego modelu.

Rodzaje zdarzeń i odbiorników

Wszystkie komponenty Swing posiadają metody `addXXXListener()` *oraz* `removeXXXListener()`, dzięki którym, we wszystkich komponentach, można dodawać i usuwać odbiorniki zdarzeń. Można również zauważyć, że w każdym przypadku `XXX` reprezentuje też argument metody, na przykład `addMyListener(MyListener m)`. Poniższa tabela zawiera związane ze sobą zdarzenia, ich odbiorniki i metody wraz z podstawowymi komponentami obsługującymi dane zdarzenie przez dostarczenie metod `addXXXListener()` i `removeXXXListener()`. Pamiętaj jednak, że projekt modelu zdarzeń umożliwia jego rozbudowę, więc można napotkać inne typy zdarzeń i odbiorników niż opisane w tabeli przedstawionej na następnej stronie.

Widać, że każdy rodzaj komponentu dostarcza jedynie pewne rodzaje zdarzeń. Okazuje się, że nie jest łatwo odszukać wszystkie zdarzenia obsługiwane przez każdy z komponentów. Łatwiejszym rozwiązaniem będzie zmodyfikowanie programu *ShowMethods.java* z rozdziału „Informacje o typach" tak, aby wyświetlał wszystkie odbiorniki zdarzeń obsługiwane przez dowolny, podany komponent Swing.

Rozdział „Informacje o typach" wprowadził technikę *refleksji*, której użyto do wyszukiwania metod podanej klasy — zarówno pełnej listy metod, jak i podzbioru takich, których nazwa pasowała do podanego słowa kluczowego. Cała „magia" refleksji polega na tym, że pokazywane są *wszystkie* metody w klasie, co uwalnia nas od chodzenia w górę hierarchii dziedziczenia i sprawdzania na każdym poziomie zawartości klas bazowych. Jest to zatem cenne narzędzie, które pozwala oszczędzić wiele czasu przy programowaniu — ponieważ nazwy większości metod Javy są jasne i samoopisujące, można wyszukać nazwy metod zawierające interesujące słowo. Kiedy już znajdzie się coś, co wydaje się odpowiednie, można sprawdzić szczegóły w dokumentacji JDK.

Oto bardziej użyteczna wersja z graficznym interfejsem, przeznaczona do wyszukiwania metod typu `addListener` w komponentach Swing:

```
// : gui/ShowAddListeners.java
// Wyświetla metody "addXXXListener" dowolnej klasy Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.regex.*;
import static net.mindview.util.SwingConsole.*;

public class ShowAddListeners extends JFrame {
  private JTextField name = new JTextField(25);
  private JTextArea results = new JTextArea(40, 65);
  private static Pattern addListener =
    Pattern.compile("(add\\w+?Listener\\(.*?\\))");
  private static Pattern qualifier =
    Pattern.compile("\\w+\\.");
  class NameL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String nm = name.getText().trim();
      if(nm.length() == 0) {
        results.setText("Brak");
        return;
```

Zdarzenie, interfejs odbiornika oraz metody dodaj-usuń	Komponenty generujące dany typ zdarzenia

`ActionEvent`
`ActionListener`
`addActionListener()`
`removeActionListener()`
	`JButton`, `JList`, `JTextField`, `JMenuItem` oraz dziedziczące po nim, wliczając w to: `JCheckBoxMenuItem`, `JMenu` oraz `JPopupMenu`.

`AdjustmentEvent`
`AdjustmentListener`
`addAdjustmentListener()`
`removeAdjustmentListener()`
	`JScrollbar` oraz wszystko, co implementuje interfejs `Adjustable`.

`ComponentEvent`
`ComponentListener`
`addComponentListener()`
`removeComponentListener()`
	*`Component` oraz dziedziczące po nim, wliczając w to: `JButton`, `JCanvas`, `JCheckBox`, `JComboBox`, `Container`, `JPanel`, `JApplet`, `JScrollPane`, `Window`, `JDialog`, `JFileDialog`, `JFrame`, `JLabel`, `JList`, `JScrollbar`, `JTextArea` i `JTextField`.

`ContainerEvent`
`ContainerListener`
`addContainerListener()`
`removeContainerListener()`
	`Container` oraz dziedziczące po nim, w tym: `JPanel`, `JApplet`, `JScrollPane`, `Window`, `JDialog`, `JFileDialog` i `JFrame`.

`FocusEvent`
`FocusListener`
`addFocusListener()`
`removeFocusListener()`
	`Component` oraz dziedziczące po nim*.

`KeyEvent`
`KeyListener`
`addKeyListener()`
`removeKeyListener()`
	`Component` oraz dziedziczące po nim*.

`MouseEvent` *(zarówno obsługa przycisków, jak i ruchu)*
`MouseListener`
`addMouseListener()`
`removeMouseListener()`
	`Component` oraz dziedziczące po nim*.

`MouseEvent`[6] *(zarówno obsługa przycisków, jak i ruchu)*
`MouseMotionListener`
`addMouseMotionListener()`
`removeMouseMotionListener()`
	`Component` oraz dziedziczące po nim*.

`WindowEvent`
`WindowListener`
`addWindowListener()`
`removeWindowListener()`
	`Window` oraz dziedziczące po nim, w tym: `JDialog`, `JFileDialog` i `JFrame`.

`ItemEvent`
`ItemListener`
`addItemListener()`
`removeItemListener()`
	`JCheckBox`, `JCheckBoxMenuItem`, `JComboBox`, `JList` oraz wszystko, co Implementuje interfejs `ItemSelectable`.

`TextEvent`
`TextListener`
`addTextListener()`
`removeTextListener()`
	Wszystko dziedziczące po `JTextComponent`, wliczając w to `JTextArea` i `JtextField`.

[6] Nic ma zdarzenia typu `MouseMotionEvent`, nawet jeżeli wydaje się, że powinno ono istnieć. Obsługa kliknięć i ruchów myszy połączona jest w `MouseEvent`, więc drugie wystąpienie tego zdarzenia w tabeli to nie błąd!

```
        }
        Class<?> kind;
        try {
          kind = Class.forName("javax.swing." + nm);
        } catch(ClassNotFoundException ex) {
          results.setText("Brak");
          return;
        }
        Method[] methods = kind.getMethods();
        results.setText("");
        for(Method m : methods) {
          Matcher matcher =
            addListener.matcher(m.toString());
          if(matcher.find())
            results.append(qualifier.matcher(
              matcher.group(1)).replaceAll("") + "\n");
        }
      }
    }
    public ShowAddListeners() {
      NameL nameListener = new NameL();
      name.addActionListener(nameListener);
      JPanel top = new JPanel();
      top.add(new JLabel("Nazwa klasy Swing (naciśnij Enter):"));
      top.add(name);
      add(BorderLayout.NORTH, top);
      add(new JScrollPane(results));
      // Początkowe dane i testy:
      name.setText("JTextArea");
      nameListener.actionPerformed(
        new ActionEvent("", 0 ,""));
    }
    public static void main(String[] args) {
      run(new ShowAddListeners(), 500, 400);
    }
  } ///:~
```

Interfejs zawiera pole tekstowe `JTextField`. Można w nim wpisać nazwę klasy Swing, która ma zostać wyświetlona. Wyniki są pobierane przy wykorzystaniu wyrażeń regularnych i wyświetlane w polu tekstowym `JTextArea`.

Jak widać, nie ma tu żadnych przycisków czy innych komponentów, przez które można by przekazać to, że chcemy rozpocząć wyszukiwanie. Jest tak dlatego, że pole `JTextField` jest monitorowane przez `ActionListener`. Kiedy po wprowadzeniu zmian zostanie naciśnięty klawisz *Enter*, lista jest natychmiast uaktualniana. Jeśli tekst nie jest pusty, zostaje użyty do wywołania `Class.forName()` w celu znalezienia klasy. Jeśli nazwa jest nieprawidłowa, wykonanie `Class.forName()` nie powiedzie się, to znaczy zostanie zgłoszony wyjątek. Jest on przechwytywany i w polu `JTextArea` jest wstawiany napis „Brak". Ale jeśli wprowadzono prawidłową nazwę (ważna jest wielkość liter), wykonanie `Class.forName()` powiedzie się i `getMethods()` zwróci tablicę obiektów typu `Method`.

W powyższym programie używane są dwa wyrażenia regularne. Pierwsze z nich — `addListener` — poszukuje łańcucha znaków „add", po którym występują dowolne znaki mogące tworzyć słowa, łańcuch „Listener" oraz lista argumentów umieszczona w nawiasach. Warto zauważyć, że całe to wyrażenie regularne zostało umieszczone w nawiasach, a zatem,

po odnalezieniu łańcucha pasującego do wyrażenia, będzie dostępne w wynikach jako „grupa". Wewnątrz metody `NameL.ActionPerformed()` tworzony jest obiekt `Matcher`; w tym celu każdy z obiektów `Method` jest przekazywany w wywołaniu metody `Pattern.matcher()`. Gdy wywołamy metodę `find()` tego obiektu, zwróci ona wartość `true` wyłącznie w przypadku odnalezienia poszukiwanego wyrażenia. W takiej sytuacji pierwszą z grup wyrażenia (zapisaną w nawiasach) można pobrać, wykorzystując wywołanie `group(1)`. Zwrócony łańcuch znaków będzie zawierać kwalifikatory; do ich usunięcia służy drugi obiekt `Pattern` — `qualifier` (jest on wykorzystywany zgodnie ze sposobem przedstawionym w programie *ShowMethods.java*).

Na końcu konstruktora wartość początkowa jest umieszczana w zmiennej `name`, a w celu przedstawienia danych początkowych zostaje zgłoszone zdarzenie.

Program ten pozwala wygodnie badać możliwości komponentu Swing. Kiedy już wiadomo, jakich zdarzeń dany komponent dostarcza, nie trzeba nigdzie szukać jego dokumentacji, aby obsłużyć zdarzenie. Wystarczy:

1. Z nazwy klasy zdarzenia usunąć słowo `Event`. Dodać słowo `Listener` do tego, co pozostało. Tak będzie nazywał się interfejs odbiornika, który trzeba zaimplementować w klasie wewnętrznej.
2. Zaimplementować powyższy interfejs, pisząc metody dla zdarzeń, które mają zostać przechwycone. Na przykład, aby obsłużyć ruch myszy, trzeba napisać kod metody `mouseMoved()` interfejsu `MouseMotionListener` (oczywiście trzeba zaimplementować również pozostałe metody, ale niebawem pokażę, że często można to uprościć).
3. Stworzyć obiekt klasy odbiornika z punktu 2. Zarejestrować go w danym komponencie przez metodę o nazwie uzyskiwanej przez dodanie przedrostka `add` do nazwy odbiornika, na przykład `addMouseMotionListener()`.

Oto niektóre z interfejsów odbiornika:

Interfejs odbiornika	Metody interfejsu
`ActionListener`	`actionPerformed(ActionEvent)`
`AdjustmentListener`	`adjustmentValueChanged(AdjustmentEvent)`

`ComponentListener`
`ComponentAdapter`
	`componentHidden(ComponentEvent)`
	`componentShown(ComponentEvent)`
	`componentMoved(ComponentEvent)`
	`componentResized(ComponentEvent)`

`ContainerListener` `ContainerAdapter`	`componentAdded(ContainerEvent)` `componentRemoved(ContainerEvent)`
`FocusListener` `FocusAdapter`	`focusGained(FocusEvent)` `focusLost(FocusEvent)`

`KeyListener`
`KeyAdapter`
	`keyPressed(KeyEvent)`
	`keyReleased(KeyEvent)`
	`keyTyped(KeyEvent)`

`MouseListener`
`MouseAdapter`
	`mouseClicked(MouseEvent)`
	`mouseEntered(MouseEvent)`
	`mouseExited(MouseEvent)`
	`mousePressed(MouseEvent)`
	`mouseReleased(MouseEvent)`

Interfejs odbiornika	Metody interfejsu
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)

Nie jest to wyczerpujący wykaz, po części dlatego, że model zdarzeń Javy pozwala na tworzenie własnych typów zdarzeń i związanych z nimi odbiorników. Bardzo często spotyka się biblioteki, które mają swoje własne zdarzenia, a wiedza zdobyta w tym rozdziale pozwoli zrozumieć, jak tych zdarzeń używać.

Uproszczone tworzenie odbiorników zdarzeń

Z powyższej tabeli wynika, że interfejsy niektórych odbiorników mają tylko jedną metodę. Są one proste do zaimplementowania. Jednak interfejsy odbiorników, które posiadają wiele metod, mogą nie być tak wygodne. Na przykład, aby przechwycić zdarzenia kliknięcia (które nie są przechwytywane na przykład przez przyciski), należy zaimplementować metodę `mouseClick()`. Jednak ponieważ `MouseListener` jest interfejsem, trzeba zaimplementować wszystkie jego pozostałe metody, nawet jeśli nic nie robią. Może to być irytujące.

Aby rozwiązać ten problem, niektóre (ale nie wszystkie) interfejsy odbiorników mające więcej niż jedną metodę są również dostarczane w postaci klas tzw. *adapterów* (ich nazwy również można znaleźć w powyższej tabeli). Dziedzicząc po adapterze, należy przesłonić jedynie te metody, które mają zostać zmienione. Na przykład adapter dla `MouseListener` jest przeważnie używany w następujący sposób:

```
class MyMouseListener extends MouseListener {
  public void mouseClicked(MouseEvent e) {
    // Obsługujemy kliknięcie ...
  }
}
```

Powodem wprowadzenia adapterów było uproszczenie tworzenia klas odbiorników.

Jednakże jest tu pewna pułapka. Przypuśćmy, że stworzymy adapter `MouseAdapter`, taki jak poniższy:

```
class MyMouseListener extends MouseAdapter {
  public void MouseClicked(MouseEvent e) {
    // Obsługujemy kliknięcie ...
  }
}
```

Nie będzie to działać, ale próba odszukania przyczyny może doprowadzić do obłędu, gdyż wszystko się kompiluje i działa prawidłowo — z wyjątkiem tego, że zamknięcie okna nie spowoduje wyjścia z programu. Czy wiesz już, w czym tkwi problem? Chodzi

o nazwę metody: `MouseClicked()` zamiast `mouseClicked()`. Drobna pomyłka w wielkości znaków prowadzi do dodania całkiem nowej metody. Jednak nie jest to metoda, która jest wywoływana przy zamykaniu okna, a więc nie otrzymamy pożądanego rezultatu. Pomimo pewnej nadmiarowości kodu interfejs zagwarantowałby, że metody zostaną prawidłowo zaimplementowane.

Pewnym sposobem zapewnienia przesłonięcia odpowiednich metod jest zastosowanie adnotacji `@Override` — wtedy w razie wpadki dowiemy się o niej od kompilatora.

Ćwiczenie 9. Bazując na programie *ShowAddListeners.java*, stwórz program posiadający wszystkie możliwości funkcjonalne programu *typeinfo/ShowMethods.java* (5).

Śledzenie wielu zdarzeń

Aby udowodnić, że te zdarzenia są rzeczywiście uruchamiane, warto stworzyć program śledzący zachowanie komponentu `JButton` (inne niż tylko to, czy przycisk został naciśnięty czy nie). Przykład ten pokazuje również, jak wyprowadzić własną klasę z klasy `JButton`[7].

Klasa `MyButton` jest klasą zagnieżdżoną w klasie `TrackEvent`, więc `MyButton` może sięgnąć do okna nadrzędnego i manipulować jego polami, co jest konieczne, aby móc wpisać informacje o stanie w pola rodzica. Oczywiście to rozwiązanie ma swoje ograniczenia, ponieważ klasa `MyButton` może być używana jedynie w połączeniu z `TrackEvent`. Tego rodzaju kod nazywa się czasami „wysoce powiązanym":

```
//: gui/TrackEvent.java
// Ujawnianie zdarzeń w czasie ich zgłaszania.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class TrackEvent extends JFrame {
  private HashMap<String,JTextField> h =
    new HashMap<String,JTextField>();
  private String[] event = {
    "focusGained", "focusLost", "keyPressed",
    "keyReleased", "keyTyped", "mouseClicked",
    "mouseEntered", "mouseExited", "mousePressed",
    "mouseReleased", "mouseDragged", "mouseMoved"
  };
  private MyButton
    b1 = new MyButton(Color.BLUE, "test1"),
    b2 = new MyButton(Color.RED, "test2");
  class MyButton extends JButton {
    void report(String field, String msg) {
      h.get(field).setText(msg);
    }
    FocusListener fl = new FocusListener() {
      public void focusGained(FocusEvent e) {
```

[7] W Java 1.0 i 1.1 dziedziczenie po obiekcie przycisku było *bezużyteczne*. Jest to kolejny z wielu fundamentalnych braków w projekcie tamtych bibliotek.

```
        report("focusGained", e.paramString());
      }
      public void focusLost(FocusEvent e) {
        report("focusLost", e.paramString());
      }
    };
    KeyListener kl = new KeyListener() {
      public void keyPressed(KeyEvent e) {
        report("keyPressed", e.paramString());
      }
      public void keyReleased(KeyEvent e) {
        report("keyReleased", e.paramString());
      }
      public void keyTyped(KeyEvent e) {
        report("keyTyped", e.paramString());
      }
    };
    MouseListener ml = new MouseListener() {
      public void mouseClicked(MouseEvent e) {
        report("mouseClicked", e.paramString());
      }
      public void mouseEntered(MouseEvent e) {
        report("mouseEntered", e.paramString());
      }
      public void mouseExited(MouseEvent e) {
        report("mouseExited", e.paramString());
      }
      public void mousePressed(MouseEvent e) {
        report("mousePressed", e.paramString());
      }
      public void mouseReleased(MouseEvent e) {
        report("mouseReleased", e.paramString());
      }
    };
    MouseMotionListener mml = new MouseMotionListener() {
      public void mouseDragged(MouseEvent e) {
        report("mouseDragged", e.paramString());
      }
      public void mouseMoved(MouseEvent e) {
        report("mouseMoved", e.paramString());
      }
    };
    public MyButton(Color color, String label) {
      super(label);
      setBackground(color);
      addFocusListener(fl);
      addKeyListener(kl);
      addMouseListener(ml);
      addMouseMotionListener(mml);
    }
  }
  public TrackEvent() {
    setLayout(new GridLayout(event.length + 1, 2));
    for(String evt : event) {
      JTextField t = new JTextField();
      t.setEditable(false);
      add(new JLabel(evt, JLabel.RIGHT));
```

```
        add(t);
        h.put(evt, t);
      }
      add(b1);
      add(b2);
    }
    public static void main(String[] args) {
      run(new TrackEvent(), 700, 500);
    }
  } ///:~
```

W konstruktorze klasy `MyButton` przez wywołanie `setBackground()` ustawiany jest kolor przycisku. Wszystkie odbiorniki zdarzeń są instalowane przez proste wywołanie odpowiedniej metody.

Klasa `TrackEvent` zawiera obiekt typu `HashMap`, służący do przechowywania łańcuchów reprezentujących typ zdarzenia oraz pola tekstowe `JTextField`, w których przechowywana jest informacja o danym zdarzeniu. Oczywiście zamiast trzymać je w `HashMap`, można by utworzyć to wszystko statycznie, ale myślę, że zgodzimy się, że ten sposób pozwala na ich łatwiejsze wykorzystanie i zmiany. Jeśli trzeba by było dodać lub usunąć jakiś typ zdarzenia w `TrackEvent`, wystarczy dodać lub usunąć odpowiedni łańcuch z tablicy zdarzeń — reszta zostanie wykonana automatycznie.

Kiedy wywoływana jest metoda `report()`, przekazywane są do niej: nazwa zdarzenia i łańcuch parametrów tego zdarzenia. Używa się przy tym odwzorowania `h` z klasy zewnętrznej, by odnaleźć pole tekstowe związane ze zdarzeniem o podanej nazwie, i wstawić łańcuch parametrów do znalezionego pola.

Dobrze jest się troszkę pobawić z tym przykładem, ponieważ można zobaczyć, o co naprawdę chodzi w zdarzeniach.

Ćwiczenie 10. Na bazie klasy `SwingConsole` napisz aplikację z przyciskiem `JButton` i polem tekstowym `JTextField`. Napisz i zarejestruj odpowiedni odbiornik zdarzeń, który po ustawieniu kursora na przycisku będzie przepisywał wpisywane znaki do pola tekstowego (6).

Ćwiczenie 11. Wyprowadź z `JButton` nowy rodzaj przycisku. Za każdym razem, kiedy ten przycisk zostanie naciśnięty, powinien zmienić swój kolor na losowo wybraną wartość. Obejrzyj program *ColorBoxes.java*, aby dowiedzieć się, jak wygenerować losowo kolor (4).

Ćwiczenie 12. Monitoruj nowy typ zdarzeń w *TrackEvent.java*, dodając nowy kod obsługi zdarzenia. Własnoręcznie musisz odszukać typ zdarzenia, które możesz monitorować (4).

Wybrane komponenty Swing

Jeśli znamy już funkcje menedżerów układu oraz model zdarzeń, to jesteśmy przygotowani na to, by dowiedzieć się, jak można używać komponentów Swing. W tej części rozdziału przedstawię przegląd najczęściej używanych komponentów i funkcji Swing.

Nie jest on w żadnej mierze wyczerpujący. Każdy przykład jest z założenia na tyle mały, by można go było łatwo przenieść i wykorzystać fragmenty jego kodu we własnych programach.

Należy pamiętać, że:

1. Bez problemów można przetestować każdy z przedstawionych przykładów, kompilując i uruchamiając pliki źródłowe rozprowadzane wraz z niniejszą książką.
2. Dokumentacja JDK, dostępna w witrynie *http://java.sun.com*, opisuje wszystkie klasy i metody komponentów Swing (tutaj pokazanych będzie tylko kilka).
3. Dzięki konwencji nazewniczej użytej dla zdarzeń Swing łatwo jest domyślić się, jak napisać i zainstalować procedurę obsługi zdarzeń danego typu. Przy poznawaniu konkretnego komponentu pomocny może być przedstawiony wcześniej program *ShowAddListeners.java*.
4. Kiedy tworzenie staje się zbyt skomplikowane, to najwyższa pora, by zacząć używać któregoś z narzędzi do tworzenia interfejsu użytkownika.

Przyciski

Biblioteka Swing zawiera kilka różnych rodzajów przycisków. Wszystkie przyciski, pola wyboru, pola opcji, a nawet elementy menu dziedziczą po klasie `AbstractButton` (choć z uwagi na to, że obejmuje ona również elementy menu, powinna być raczej nazwana bardziej ogólnie, np. „AbstractSelector"). Niebawem pokażę, jak używać menu. Tymczasem poniższy przykład prezentuje różne rodzaje dostępnych przycisków:

```
//: gui/Buttons.java
// Różne przyciski biblioteki Swing.
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.basic.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Buttons extends JFrame {
  private JButton jb = new JButton("JButton");
  private BasicArrowButton
    up = new BasicArrowButton(BasicArrowButton.NORTH),
    down = new BasicArrowButton(BasicArrowButton.SOUTH),
    right = new BasicArrowButton(BasicArrowButton.EAST),
    left = new BasicArrowButton(BasicArrowButton.WEST);
  public Buttons() {
    setLayout(new FlowLayout());
    add(jb);
    add(new JToggleButton("JToggleButton"));
    add(new JCheckBox("JCheckBox"));
    add(new JRadioButton("JRadioButton"));
    JPanel jp = new JPanel();
    jp.setBorder(new TitledBorder("Directions"));
    jp.add(up);
    jp.add(down);
    jp.add(left);
    jp.add(right);
    add(jp);
```

```
  }
  public static void main(String[] args) {
    run(new Buttons(), 350, 200);
  }
} ///:~
```

Przykład prezentuje różne, specyficzne typy przycisków, poczynając od przycisku typu `BasicArrowButton` z `javax.swing.plaf.basic`. Jeśli uruchomimy program, zorientujemy się, że przycisk przełączalny (`JToggleButton`) utrzymuje swoją ostatnią pozycję — jest wciśnięty lub wyciśnięty. Natomiast przycisk wyboru (`JRadioButton`) i pole opcji (`JCheckBox`) zachowują się tak samo — można je włączać i wyłączać (obydwa dziedziczą po `JToggleButton`).

Grupy przycisków

Jeśli chcemy, by przełączanie przycisków wyboru odbywało się na zasadzie „alternatywy wykluczającej", musimy je umieścić w jednej „grupie przycisków". Jednak, jak demonstruje to poniższy przykład, do grupy przycisków `ButtonGroup` dodać można dowolny przycisk dziedziczący po typie `AbstractButton`.

Aby uniknąć powtarzania kodu w przykładzie, używam refleksji, aby stworzyć grupy różnego rodzaju przycisków. Widać to w metodzie `makeBPanel()`, która tworzy grupę przycisków i panel. Drugim parametrem `makeBPanel()` jest tablica etykiet. Dla każdej etykietki z tej tablicy do panelu dodawany jest przycisk klasy podanej w pierwszym parametrze.

```
//: gui/ButtonGroups.java
// Zastosowanie refleksji do tworzenia grup
// przycisków różnych podtypów AbstractButton.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.lang.reflect.*;
import static net.mindview.util.SwingConsole.*;

public class ButtonGroups extends JFrame {
  private static String[] ids = {
    "Jurek", "Wit", "Tomek", "Waldek", "Edek", "Wacek"
  };
  static JPanel makeBPanel(
    Class<? extends AbstractButton> kind, String[] ids) {
    ButtonGroup bg = new ButtonGroup();
    JPanel jp = new JPanel();
    String title = kind.getName();
    title = title.substring(title.lastIndexOf('.') + 1);
    jp.setBorder(new TitledBorder(title));
    for(String id : ids) {
      AbstractButton ab = new JButton("porażka");
      try {
        // Pobranie metody konstrukcji
        // przyjmującej argument String:
        Constructor ctor =
          kind.getConstructor(String.class);
        // Utworzenie nowego obiektu:
        ab = (AbstractButton)ctor.newInstance(id);
```

```
        } catch(Exception ex) {
          System.err.println("nie można utworzyć " + kind);
        }
        bg.add(ab);
        jp.add(ab);
      }
      return jp;
    }
    public ButtonGroups() {
      setLayout(new FlowLayout());
      add(makeBPanel(JButton.class, ids));
      add(makeBPanel(JToggleButton.class, ids));
      add(makeBPanel(JCheckBox.class, ids));
      add(makeBPanel(JRadioButton.class, ids));
    }
    public static void main(String[] args) {
      run(new ButtonGroups(), 500, 350);
    }
} ///:~
```

Tytuł ramki jest brany z nazwy klasy, a wszystkie informacje o ścieżce są uprzednio usuwane. Referencja typu `AbstractButton` jest inicjalizowana na obiekt `JButton` z etykietą „porażka", więc nawet jeśli zignorujemy wyjątek, to nadal na ekranie będzie widoczny problem. Metoda `getConstructor()` przyjmuje tablicę typów parametrów i tworzy odpowiadający takiej sygnaturze obiekt `Constructor`. Wtedy wystarczy wywołać metodę `newInstance()`, przekazując jej tablicę obiektów zawierających rzeczywiste parametry konstruktora — w tym przypadku etykiety z tablicy `ids`.

Aby uzyskać działanie przycisków na zasadzie „alternatywy wykluczającej", wystarczy stworzyć grupę przycisków i dodać do niej wszystkie przyciski, które mają tak działać. Po uruchomieniu programu można się przekonać, że wszystkie przyciski poza `JButton` zachowują się wzajemnie wykluczająco.

Ikony

Ikon można używać w etykietach `JLabel` lub w czymkolwiek, co dziedziczy po typie `AbstractButton` (włączając przyciski typu `JButton`, `JCheckBox`, `JRadioButton` i wszelkie rodzaje elementów menu `JMenuItem`). Użycie ikon w etykietach `JLabel` jest bardzo proste (później zostanie przedstawiony odpowiedni przykład). Poniższy przykład przedstawia wszystkie dodatkowe sposoby użycia ikon w przyciskach i ich pochodnych.

Można użyć dowolnych plików w formacie GIF. Pliki używane w tym przykładzie są częścią kodu rozpowszechnianego z książką, dostępnego pod adresem *www.MindView.net*. Aby otworzyć plik i wczytać obrazek, wystarczy stworzyć obiekt typu `ImageIcon` i przekazać mu nazwę pliku. W wyniku otrzymujemy ikonę, którą można użyć dalej w programie.

```
//: gui/Faces.java
// Ikony na przyciskach.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;
```

```
public class Faces extends JFrame {
  private static Icon[] faces;
  private JButton jb, jb2 = new JButton("Wyłącz");
  private boolean mad = false;
  public Faces() {
    faces = new Icon[]{
      new ImageIcon(getClass().getResource("Face0.gif")),
      new ImageIcon(getClass().getResource("Face1.gif")),
      new ImageIcon(getClass().getResource("Face2.gif")),
      new ImageIcon(getClass().getResource("Face3.gif")),
      new ImageIcon(getClass().getResource("Face4.gif")),
    };
    jb = new JButton("JButton", faces[3]);
    setLayout(new FlowLayout());
    jb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if(mad) {
          jb.setIcon(faces[3]);
          mad = false;
        } else {
          jb.setIcon(faces[0]);
          mad = true;
        }
        jb.setVerticalAlignment(JButton.TOP);
        jb.setHorizontalAlignment(JButton.LEFT);
      }
    });
    jb.setRolloverEnabled(true);
    jb.setRolloverIcon(faces[1]);
    jb.setPressedIcon(faces[2]);
    jb.setDisabledIcon(faces[4]);
    jb.setToolTipText("Hej!");
    add(jb);
    jb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if(jb.isEnabled()) {
          jb.setEnabled(false);
          jb2.setText("Włącz");
        } else {
          jb.setEnabled(true);
          jb2.setText("Wyłącz");
        }
      }
    });
    add(jb2);
  }
  public static void main(String[] args) {
    run(new Faces(), 250, 125);
  }
} ///:~
```

Ikon można używać w wielu konstruktorach, natomiast by ustawić lub zmienić obrazek ikony, można również użyć metody `setIcon()`. Powyższy przykład pokazuje też, jak przycisk `JButton` (i każdy inny obiekt podobny do przycisku dziedziczący po `AbstractButton`) może wyświetlać różne ikony w odpowiedzi na zdarzenia związane z tym przyciskiem: kiedy jest wciskany, wyłączany lub kiedy przejedzie nad nim kursor myszy (ang. *roll over*). Powoduje to, że przycisk wydaje się bardziej interaktywny.

Podpowiedzi

W ostatnim przykładzie do przycisku została dodana „podpowiedź". Prawie wszystkie klasy używane przy tworzeniu interfejsu dziedziczą po klasie JComponent, która zawiera metodę o nazwie setToolTipText(String). Zatem dla każdego obiektu, który może zostać umieszczony na formatce, wystarczy napisać (dla obiektu jc dowolnej klasy dziedziczącej po JComponent):

```
jc.setToolTipText("Moja podpowiedź");
```

Gdy mysz pozostanie nad tym komponentem przez ustalony czas, obok jej kursora pojawi się małe okno zawierające podany tekst.

Pola tekstowe

Poniższy przykład prezentuje dodatkowe funkcje pól tekstowych JTextField:

```
//: gui/TextFields.java
// Pola tekstowe a zdarzenia Javy.
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class TextFields extends JFrame {
  private JButton
    b1 = new JButton("Odczytaj tekst"),
    b2 = new JButton("Wpisz tekst");
  private JTextField
    t1 = new JTextField(30),
    t2 = new JTextField(30),
    t3 = new JTextField(30);
  private String s = "";
  private UpperCaseDocument ucd = new UpperCaseDocument();
  public TextFields() {
    t1.setDocument(ucd);
    ucd.addDocumentListener(new T1());
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    t1.addActionListener(new T1A());
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
    add(t1);
    add(t2);
    add(t3);
  }
  class T1 implements DocumentListener {
    public void changedUpdate(DocumentEvent e) {}
    public void insertUpdate(DocumentEvent e) {
      t2.setText(t1.getText());
      t3.setText("Tekst: "+ t1.getText());
    }
```

```
    public void removeUpdate(DocumentEvent e) {
      t2.setText(t1.getText());
    }
  }
  class T1A implements ActionListener {
    private int count = 0;
    public void actionPerformed(ActionEvent e) {
      t3.setText("t1 ActionEvent " + count++);
    }
  }
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(t1.getSelectedText() == null)
        s = t1.getText();
      else
        s = t1.getSelectedText();
      t1.setEditable(true);
    }
  }
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      ucd.setUpperCase(false);
      t1.setText("Wstawiony przez Przycisk 2: " + s);
      ucd.setUpperCase(true);
      t1.setEditable(false);
    }
  }
  public static void main(String[] args) {
    run(new TextFields(), 375, 200);
  }
}

class UpperCaseDocument extends PlainDocument {
  private boolean upperCase = true;
  public void setUpperCase(boolean flag) {
    upperCase = flag;
  }
  public void
  insertString(int offset, String str, AttributeSet attSet)
  throws BadLocationException {
    if(upperCase) str = str.toUpperCase();
    super.insertString(offset, str, attSet);
  }
} ///:~
```

Pole tekstowe `t3` służy do powiadamiania o tym, że uruchomiony został odbiornik zdarzeń pola tekstowego `t1`. Dzięki temu można się przekonać, że odbiornik zdarzeń dla pól tekstowych `JTextField` jest uruchamiany dopiero po naciśnięciu klawisza *Enter*.

Do pola tekstowego `t1` doczepionych jest kilka odbiorników zdarzeń. Odbiornik `T1` jest typu `DocumentListener` i odpowiada na każde zmiany w „dokumencie" (w tym przypadku chodzi o zawartość pola `JTextField`). Kopiuje on automatycznie cały tekst z `t1` do `t2`. Dodatkowo dokument pola `t1` jest ustawiany na klasę o nazwie `UpperCaseDocument`, dziedziczącą po typie `PlainDocument`, przez co wszystkie znaki zamieniane są na wielkie litery. Pole tekstowe automatycznie wykrywa znaki kasujące (backspace) i dokonuje usunięcia, przesuwa kursor i obsługuje wszystko, czego można w tej sytuacji oczekiwać.

Ćwiczenie 13. Zmodyfikuj program *TextFields.java* tak, aby znaki wpisywane w polu t2 zachowywały oryginalną wielkość, to znaczy nie były automatycznie zamieniane na wielkie litery (3).

Ramki

Klasa JComponent zawiera metodę o nazwie setBorder(), pozwalającą na ustawienie dla każdego komponentu interesującej ramki. Poniższy przykład prezentuje kilka dostępnych ramek. Używa w tym celu metody o nazwie showBorder(), która tworzy panel JPanel i ustawia mu odpowiednią ramkę. Do ustalenia nazwy podanej ramki używany jest mechanizm RTTI (usuwający przy okazji każdą informację o pakiecie, z którego pochodzi klasa). Uzyskana w ten sposób nazwa umieszczana jest na środku panelu w etykiecie JLabel:

```
//: gui/Borders.java
// Różne ramki Swing.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Borders extends JFrame {
  static JPanel showBorder(Border b) {
    JPanel jp = new JPanel();
    jp.setLayout(new BorderLayout());
    String nm = b.getClass().toString();
    nm = nm.substring(nm.lastIndexOf('.') + 1);
    jp.add(new JLabel(nm, JLabel.CENTER),
      BorderLayout.CENTER);
    jp.setBorder(b);
    return jp;
  }
  public Borders() {
    setLayout(new GridLayout(2,4));
    add(showBorder(new TitledBorder("Tytuł")));
    add(showBorder(new EtchedBorder()));
    add(showBorder(new LineBorder(Color.BLUE)));
    add(showBorder(
      new MatteBorder(5,5,30,30,Color.GREEN)));
    add(showBorder(
      new BevelBorder(BevelBorder.RAISED)));
    add(showBorder(
      new SoftBevelBorder(BevelBorder.LOWERED)));
    add(showBorder(new CompoundBorder(
      new EtchedBorder(),
      new LineBorder(Color.RED))));
  }
  public static void main(String[] args) {
    run(new Borders(), 500, 300);
  }
} ///:~
```

Można również tworzyć własne ramki i umieszczać je wewnątrz przycisków, etykiet itd. — wszystkiego, co dziedziczy po JComponent.

Miniedytor

Używając kontrolki JTextPane, otrzymujemy bez wielkiego wysiłku ogromne możliwości edycji tekstu. Poniższy przykład używa jej w bardzo prosty sposób, ignorując większość jej funkcji:

```
//: gui/TextPane.java
// Kontrolka JTextPane w roli prostego edytora.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

public class TextPane extends JFrame {
  private JButton b = new JButton("Dodaj Tekst");
  private JTextPane tp = new JTextPane();
  private static Generator sg =
    new RandomGenerator.String(7);
  public TextPane() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(int i = 1; i < 10; i++)
          tp.setText(tp.getText() + sg.next() + "\n");
      }
    });
    add(new JScrollPane(tp));
    add(BorderLayout.SOUTH, b);
  }
  public static void main(String[] args) {
    run(new TextPane(), 475, 425);
  }
} ///:~
```

Przycisk dodaje losowo wygenerowany tekst. Panel JTextPane ma służyć do edycji tekstu na miejscu, więc nie widzimy tu nigdzie metody `append()`. W tym przypadku (przyznaję, że jest to marne użycie możliwości panelu JTextPane) tekst musi zostać przechwycony, zmodyfikowany i umieszczony ponownie na panelu przez wywołanie metody `setText()`.

Jak wcześniej powiedziałem, domyślnie przyjmowanym przez ramkę menedżerem układu jest `BorderLayout`. W ramce osadzany jest element `JTextPane` (w `JScrollPane`), a skoro nie są podawane żadne rozmiary, element wypełni cały środek ramki. Przycisk (`JButton`) dodawany jest na konkretnym obszarze (`SOUTH`) i wypełnia sobą ten obszar — w tym przypadku przycisk znajdzie się na spodzie ekranu.

Proszę zwrócić uwagę na wbudowane funkcje panelu JTextPane, takie jak np. automatyczne zawijanie wierszy. Posiada on wiele innych ciekawych funkcji, które można wyszukać w dokumentacji JDK.

Ćwiczenie 14. Zmodyfikuj program *TextPane.java* tak, by używał komponentu JTextArea zamiast JTextPane (2).

Pola wyboru

Pole wyboru (ang. *check box*) pozwala na dokonywanie pojedynczych wyborów tak-nie. Składa się z małego kwadratu i etykietki. Kwadrat zawiera najczęściej mały znaczek x (lub jakiś inny znacznik) lub jest pusty w zależności od tego, czy to pole jest wybrane.

Normalnie tworzy się pole JCheckBox, używając konstruktora przyjmującego jako parametr żądaną etykietę. Po utworzeniu pola opcji można odczytywać i zmieniać jego stan, jak również odczytywać i zmieniać wyświetlaną etykietę.

Za każdym razem kiedy pole opcji JCheckBox jest ustawiane bądź czyszczone, zachodzi zdarzenie. Można je przechwycić w taki sam sposób, w jaki robi się to z przyciskami, tj. przez ustawienie odbiornika tego zdarzenia. Poniższy przykład używa pola tekstowego JTextArea do wyliczenia wszystkich zaznaczonych pól:

```
//: gui/CheckBoxes.java
// Stosowanie pól wyboru
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class CheckBoxes extends JFrame {
  private JTextArea t = new JTextArea(6, 15);
  private JCheckBox
    cb1 = new JCheckBox("Pole wyboru 1"),
    cb2 = new JCheckBox("Pole wyboru 2"),
    cb3 = new JCheckBox("Pole wyboru 3");
  public CheckBoxes() {
    cb1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        trace("1", cb1);
      }
    });
    cb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        trace("2", cb2);
      }
    });
    cb3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        trace("3", cb3);
      }
    });
    setLayout(new FlowLayout());
    add(new JScrollPane(t));
    add(cb1);
    add(cb2);
    add(cb3);
  }
  private void trace(String b, JCheckBox cb) {
    if(cb.isSelected())
      t.append("Pole " + b + " ustawione\n");
    else
      t.append("Pole " + b + " wyczyszczone\n");
  }
```

```
  public static void main(String[] args) {
    run(new CheckBoxes(), 200, 300);
  }
} ///:~
```

Metoda `trace()` wysyła do pola tekstowego nazwę i aktualny stan wybranego pola `JCheckBox`, używając metody `append()`. W ten sposób otrzymujemy listę wybieranych pól i ich kolejnych stanów.

Ćwiczenie 15. Dodaj do programu stworzonego w ćwiczeniu 5. pole opcji wyboru (`JCheckBox`), przechwyć zdarzenie i wstaw jakiś tekst do pola tekstowego (5).

Przyciski wyboru

Pomysł przycisków wyboru (ang. *radio button*) w graficznych interfejsach użytkownika pochodzi od mechanicznych przycisków w starych radiach samochodowych: kiedy naciśnęło się jeden z przycisków, wyskakiwał przycisk wciśnięty wcześniej. Pozwalało to wymusić dokonywanie pojedynczego wyboru spomiędzy wielu dostępnych opcji.

Aby utworzyć grupę powiązanych ze sobą przycisków `JRadioButton`, wystarczy umieścić je w grupie `ButtonGroup` (na jednej formatce może występować dowolna liczba grup przycisków). Opcjonalnie jeden z przycisków może mieć ustawiony stan początkowy na `true` (używa się do tego drugiego parametru konstruktora). Jeśli spróbujemy ustawić kilka przycisków wyboru, to tylko ostatni ustawiany będzie zaznaczony.

Oto przykład użycia przycisków wyboru z przechwytywaniem zdarzeń za pomocą odbiornika `ActionListener`:

```
//: gui/RadioButtons.java
// Przyciski JRadioButton.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class RadioButtons extends JFrame {
  private JTextField t = new JTextField(15);
  private ButtonGroup g = new ButtonGroup();
  private JRadioButton
    rb1 = new JRadioButton("jeden", false),
    rb2 = new JRadioButton("dwa", false),
    rb3 = new JRadioButton("trzy", false);
  private ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      t.setText("Przycisk wyboru " +
        ((JRadioButton)e.getSource()).getText());
    }
  };
  public RadioButtons() {
    rb1.addActionListener(al);
    rb2.addActionListener(al);
    rb3.addActionListener(al);
    g.add(rb1); g.add(rb2); g.add(rb3);
    t.setEditable(false);
```

```
    setLayout(new FlowLayout());
    add(t);
    add(rb1);
    add(rb2);
    add(rb3);
  }
  public static void main(String[] args) {
    run(new RadioButtons(), 200, 125);
  }
} ///:~
```

Do wyświetlania stanu używane jest zwykłe pole tekstowe. Pole jest nieedytowalne, ponieważ wykorzystuje się je wyłącznie do wyświetlania danych, a nie do ich pobierania. Zamiast tego można było użyć tutaj etykiety `JLabel`.

Listy rozwijane

Listy rozwijane (ang. *drop-down list, combo box*), podobnie jak grupy przycisków wyboru, pozwalają wymusić na użytkowniku wybór tylko jednego elementu z grupy różnych możliwości. Jest to jednak bardziej poręczne rozwiązanie i łatwiej zmienić elementy listy, nie zaskakując użytkownika (można dynamicznie zmieniać przyciski wyboru, ale najczęściej jest to wizualnie rażące).

Domyślnie lista rozwijana w Javie nie działa tak samo, jak listy rozwijane w Windows, które pozwalają na wybranie elementu z listy *lub* wprowadzenie własnego. Aby lista rozwijana działała w sposób znany z systemu Windows, należy wywołać jej metodę `setEditable()`. W kontrolce `JComboBox` można wybrać jeden i tylko jeden z elementów listy. W poniższym przykładzie zaraz po uruchomieniu kontrolka `JComboBox` zawiera pewną liczbę elementów, a kolejne są dodawane dopiero po naciśnięciu przycisku.

```
//: gui/ComboBoxes.java
// Listy rozwijane.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class ComboBoxes extends JFrame {
  private String[] description = {
    "Rozentuzjazmowany", "Ograniczony", "Krnąbrny", "Błyskotliwy",
    "Samolubny", "Nieśmiały", "Zdrowy", "Zgniły"
  };
  private JTextField t = new JTextField(15);
  private JComboBox c = new JComboBox();
  private JButton b = new JButton("Dodaj pozycję");
  private int count = 0;
  public ComboBoxes() {
    for(int i = 0; i < 4; i++)
      c.addItem(description[count++]);
    t.setEditable(false);
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if(count < description.length)
          c.addItem(description[count++]);
      }
```

```
    });
    c.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        t.setText("indeks: "+ c.getSelectedIndex() + "    " +
         ((JComboBox)e.getSource()).getSelectedItem());
      }
    });
    setLayout(new FlowLayout());
    add(t);
    add(c);
    add(b);
  }
  public static void main(String[] args) {
    run(new ComboBoxes(), 200, 175);
  }
} ///:~
```

Pole tekstowe wyświetla indeks wybranego elementu, czyli numer aktualnie wybranego elementu w sekwencji oraz etykietkę obok przycisku.

Listy

Pola listy znacząco różnią się od list rozwijanych `JComboBox` i to nie tylko wyglądem. Lista typu `JComboBox` rozwija się dopiero przy aktywacji. Natomiast lista typu `JList` zawsze zajmuje z góry określony obszar ekranu. Aby otrzymać aktualnie zaznaczone elementy listy, wystarczy wywołać metodę `getSelectedValues()`, która zwraca tablicę etykiet wybranych elementów.

Lista `JList` pozwala na dokonywanie wielokrotnego wyboru: klikając myszą więcej niż jeden element i jednocześnie trzymając wciśnięty klawisz *Ctrl*, pozostawiamy elementy już wybrane zaznaczone i możemy dodać tyle nowych, ile tylko chcemy. Jeśli natomiast po wybraniu jednego elementu kliknie się drugi, trzymając jednocześnie wciśnięty klawisz *Shift*, to zostaną zaznaczone wszystkie elementy znajdujące się pomiędzy tymi dwoma wybranymi. Aby usunąć element z wybranej grupy, wystarczy ponownie go kliknąć, trzymając *Ctrl*.

```
//: gui/List.java
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class List extends JFrame {
  private String[] flavors = {
    "Czekoladowe", "Truskawkowe", "Waniliowe",
    "Miętowe", "Mocca", "Rumowe",
    "Karmelowe", "Bakaliowe"
  };
  private DefaultListModel lItems = new DefaultListModel();
  private JList lst = new JList(lItems);
  private JTextArea t =
    new JTextArea(flavors.length, 20);
```

```
  private JButton b = new JButton("Dodaj pozycję");
  private ActionListener bl = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      if(count < flavors.length) {
        llItems.add(0, flavors[count++]);
      } else {
        // Wyłączmy, bo nie ma już smaków,
        // które można by dodać do listy
        b.setEnabled(false);
      }
    }
  };
  private ListSelectionListener ll =
    new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent e) {
        if(e.getValueIsAdjusting()) return;
        t.setText("");
        for(Object item : lst.getSelectedValues())
          t.append(item + "\n");
      }
    };
  private int count = 0;
  public List() {
    t.setEditable(false);
    setLayout(new FlowLayout());
    // Utworzenie ramek dla komponentów:
    Border brd = BorderFactory.createMatteBorder(
      1, 1, 2, 2, Color.BLACK);
    lst.setBorder(brd);
    t.setBorder(brd);
    // Dodajmy do listy pierwsze cztery pozycje
    for(int i = 0; i < 4; i++)
      llItems.addElement(flavors[count++]);
    add(t);
    add(lst);
    add(b);
    // Rejestracja odbiorników zdarzeń
    lst.addListSelectionListener(ll);
    b.addActionListener(bl);
  }
  public static void main(String[] args) {
    run(new List(), 250, 375);
  }
} ///:~
```

Jak widać, tutaj również do listy dodano ramkę.

Jeśli chcemy do listy `JList` wstawić całą tablicę napisów, możemy wykorzystać bardzo prostą metodę: wystarczy przekazać tę tablicę konstruktorowi, a on już automatycznie zbuduje listę. Jedynym powodem używania „modelu listy" w powyższym przykładzie jest dodanie możliwości operacji na liście w trakcie działania programu.

Kontrolka `JList` nie obsługuje suwaków automatycznie. Oczywiście wystarczy tylko opakować ją panelem `JScrollPane`, a wszystkie detale zostaną obsłużone automatycznie.

Ćwiczenie 16. Uprość program *List.java*, przekazując do konstruktora tablicę i eliminując dynamiczne dodawanie elementów do listy (5).

Zakładki

Kontrolka JTabbedPane pozwala na tworzenie „dialogów z zakładkami". Posiadają one zakładki podobne do tych, jakie stosuje się do porządkowania w szufladach teczek z aktami. Naciskanie klawisza *Tab* przełącza widok na kolejne zakładki.

```
//: gui/TabbedPane1.java
// Zakładki.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class TabbedPane1 extends JFrame {
  private String[] flavors = {
    "Czekoladowe", "Truskawkowe", "Waniliowe",
    "Miętowe", "Mocca", "Rumowe",
    "Karmelowe", "Bakaliowe"
  };
  private JTabbedPane tabs = new JTabbedPane();
  private JTextField txt = new JTextField(20);
  public TabbedPane1() {
    int i = 0;
    for(String flavor : flavors)
      tabs.addTab(flavors[i],
        new JButton("Panel zakładki " + i++));
    tabs.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        txt.setText("Wybrana zakładka: " +
          tabs.getSelectedIndex());
      }
    });
    add(BorderLayout.SOUTH, txt);
    add(tabs);
  }
  public static void main(String[] args) {
    run(new TabbedPane1(), 400, 250);
  }
} ///:~
```

Po uruchomieniu programu można zaobserwować, że panel JTabbedPane automatycznie układa zakładki w stos, jeśli jest ich zbyt wiele, by mogły się zmieścić w jednym wierszu. Można to sprawdzić, uruchamiając program z wiersza poleceń i zmieniając rozmiar okna.

Okna komunikatów

Środowiska okienkowe często zawierają standardowy zestaw okien komunikatów, które pozwalają na szybkie przekazanie informacji użytkownikowi lub też pobranie informacji od użytkownika. W Swingu okna te są dostępne poprzez komponent JOptionPane. Mamy wiele różnych możliwości (niektóre bardzo wyszukane), ale najczęściej używanymi będą: okno wiadomości oraz okno potwierdzenia, wywoływane przez statyczne metody JOptionPane.showMessageDialog() i JOptionPane.showConfirmDialog(). Poniższy przykład pokazuje część okien dialogowych dostępnych poprzez JOptionPane:

```
//: gui/MessageBoxes.java
// Możliwości kontrolek JOptionPane.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class MessageBoxes extends JFrame {
  private JButton[] b = {
    new JButton("Uwaga"), new JButton("Tak-Nie"),
    new JButton("Kolor"), new JButton("Wejście"),
    new JButton("3-wartościowe")
  };
  private JTextField txt = new JTextField(15);
  private ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      String id = ((JButton)e.getSource()).getText();
      if(id.equals("Uwaga"))
        JOptionPane.showMessageDialog(null,
          "Masz robaka we włosach!", "Hej!",
          JOptionPane.ERROR_MESSAGE);
      else if(id.equals("Tak-Nie"))
        JOptionPane.showConfirmDialog(null,
          "albo nie", "wybierz tak",
          JOptionPane.YES_NO_OPTION);
      else if(id.equals("Kolor")) {
        Object[] options = { "Czerwony", "Zielony" };
        int sel = JOptionPane.showOptionDialog(
          null, "Wybierz kolor!", "Ostrzeżenie",
          JOptionPane.DEFAULT_OPTION,
          JOptionPane.WARNING_MESSAGE, null,
          options, options[0]);
        if(sel != JOptionPane.CLOSED_OPTION)
          txt.setText("Wybrany kolor: " + options[sel]);
      } else if(id.equals("Wejście")) {
        String val = JOptionPane.showInputDialog(
            "Ile widzisz palców?");
        txt.setText(val);
      } else if(id.equals("3-wartościowe")) {
        Object[] selections = {"Pierwszy", "Drugi", "Trzeci"};
        Object val = JOptionPane.showInputDialog(
          null, "Wybierz jedno", "Wejście",
          JOptionPane.INFORMATION_MESSAGE,
          null, selections, selections[0]);
        if(val != null)
          txt.setText(val.toString());
      }
    }
  };
  public MessageBoxes() {
    setLayout(new FlowLayout());
    for(int i = 0; i < b.length; i++) {
      b[i].addActionListener(al);
      add(b[i]);
    }
    add(txt);
  }
```

```
  public static void main(String[] args) {
    run(new MessageBoxes(), 200, 200);
  }
} ///:~
```

Aby ograniczyć się tylko do jednego odbiornika zdarzeń, zastosowałem trochę ryzykowne podejście — sprawdzanie etykiet na przyciskach. Problemem jest to, że pisząc nazwę, łatwo można się pomylić (najczęściej co do wielkości znaków), a taki błąd jest trudny do wykrycia.

Zauważ, że metoda `showOptionDialog()` i `showInputDialog()` zwracają obiekty zawierające wartość wprowadzoną przez użytkownika.

Ćwiczenie 17. Utwórz aplikację, ponownie używając klasy `SwingConsole`. W dokumentacji JDK z *java.sun.com* odszukaj opis komponentu `JPasswordField` i umieść go w programie. Jeśli użytkownik wprowadzi prawidłowe hasło, wyświetl potwierdzenie, używając okienka `JOptionPane` (5).

Ćwiczenie 18. Zmodyfikuj program *MessageBoxes.java* tak, aby zawierał odrębny odbiornik `ActionListener` dla każdego przycisku (zamiast sprawdzania etykiety przycisku) (4).

Menu

Każdy komponent zdolny do wyświetlania menu, wliczając w to: `JApplet`, `JFrame`, `JDialog` i ich pochodne, zawiera metodę `setJMenuBar()`, która przyjmuje obiekt `JMenuBar` (przy czym każdy komponent może mieć tylko jedno menu). Na pasku menu `JMenuBar` można umieszczać kolejne menu, dodając obiekty typu `JMenu`. Do nich z kolei można dodać pozycje, wstawiając obiekty typu `JMenuItem`. Każdy taki element może mieć podpięty własny odbiornik zdarzeń `ActionListener`, który będzie uruchamiany, jeśli wybrany zostanie właśnie ten element menu.

W Javie i Swingu trzeba ręcznie układać wszystkie menu w kodzie źródłowym. Oto przykład bardzo prostego menu:

```
//: gui/SimpleMenus.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class SimpleMenus extends JFrame {
  private JTextField t = new JTextField(15);
  private ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      t.setText(((JMenuItem)e.getSource()).getText());
    }
  };
  private JMenu[] menus = {
    new JMenu("Ene"), new JMenu("Due"),
    new JMenu("Rabe")
  };
  private JMenuItem[] items = {
    new JMenuItem("Złapał"), new JMenuItem("Bocian"),
```

```
    new JMenuItem("Żabę"), new JMenuItem("A"),
    new JMenuItem("Żaba"), new JMenuItem("Chińczyka"),
    new JMenuItem("Co"), new JMenuItem("Z tego"),
    new JMenuItem("Wynika")
  };
  public SimpleMenus() {
    for(int i = 0; i < items.length; i++) {
      items[i].addActionListener(al);
      menus[i % 3].add(items[i]);
    }
    JMenuBar mb = new JMenuBar();
    for(JMenu jm : menus)
      mb.add(jm);
    setJMenuBar(mb);
    setLayout(new FlowLayout());
    add(t);
  }
  public static void main(String[] args) {
    run(new SimpleMenus(), 200, 150);
  }
} ///:~
```

Używając operatora dzielenia modulo `i%3`, rozdzieliłem elementy pomiędzy trzy menu. Każda pozycja menu musi mieć określony odbiornik zdarzeń — tutaj dla wszystkich używany jest ten sam, ale zazwyczaj będziesz musiał dla każdego elementu określić odrębny odbiornik zdarzeń.

Komponent `JMenuItem` dziedziczy po `AbstractButton`, dlatego można go używać podobnie jak przycisków. Sam w sobie jest elementem, który można umieszczać w rozwijanym menu. Z `JMenuItem` wywodzą się trzy kolejne typy: `JMenu` przechowujący elementy `JMenuItem` (w ten sposób można otrzymać menu kaskadowe), element `JCheckBoxMenuItem` mogący dodatkowo wyświetlać znaczek oznaczający, czy dana pozycja menu jest aktualnie wybrana, oraz element `JRadioButtonMenuItem` pozwalający uzyskać przyciski wyboru.

Ponownie wykorzystam „smaki lodów", tym razem w bardziej wyrafinowanym przykładzie tworzenia menu. Ten przykład pokazuje też menu kaskadowe, skróty klawiaturowe, pozycje `JCheckBoxMenuItem` oraz sposób na dynamiczne zmiany menu:

```
// : gui/Menus.java
// Menu kaskadowe, pola wyboru w menu, podmienianie
// pozycji menu, skróty klawiaturowe i polecenia akcji.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Menus extends JFrame {
  private String[] flavors = {
    "Czekoladowe", "Truskawkowe", "Waniliowe",
    "Miętowe", "Mocca", "Rumowe",
    "Karmelowe", "Bakaliowe"
  };
  private JTextField t = new JTextField("Brak smaku", 30);
  private JMenuBar mbl = new JMenuBar();
  private JMenu
    f = new JMenu("Plik"),
```

```
    m = new JMenu("Smaki");
    s = new JMenu("Bezpieczeństwo");
  // Albo tak:
  private JCheckBoxMenuItem[] safety = {
    new JCheckBoxMenuItem("Chroń"),
    new JCheckBoxMenuItem("Ukryj")
  };
  private JMenuItem[] file = { new JMenuItem("Otwórz") };
  // Podmiana drugiego paska menu:
  private JMenuBar mb2 = new JMenuBar();
  private JMenu fooBar = new JMenu("blaBla");
  private JMenuItem[] other = {
    // Dodawanie skrótu klawiaturowego jest bardzo proste,
    // ale można go dodać do pozycji menu (JMenuItem) tylko
    // w konstruktorze pozycji:
    new JMenuItem("Ble", KeyEvent.VK_F),
    new JMenuItem("Bla", KeyEvent.VK_A),
    // Bez skrótu klawiaturowego:
    new JMenuItem("Ple"),
  };
  private JButton b = new JButton("Wymiana Menu");
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuBar m = getJMenuBar();
      setJMenuBar(m == mb1 ? mb2 : mb1);
      validate(); // Odświeżenie ramki
    }
  }
  class ML implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuItem target = (JMenuItem)e.getSource();
      String actionCommand = target.getActionCommand();
      if(actionCommand.equals("Otwórz")) {
        String s = t.getText();
        boolean chosen = false;
        for(String flavor : flavors)
          if(s.equals(flavor))
            chosen = true;
        if(!chosen)
          t.setText("Najpierw wybierz smak!");
        else
          t.setText("Otwieram " + s + ". Mmm, mm!");
      }
    }
  }
  class FL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuItem target = (JMenuItem)e.getSource();
      t.setText(target.getText());
    }
  }
  // Alternatywnie można utworzyć oddzielną klasę dla każdego
  // osobnego elementu MenuItem. Potem nie trzeba się przejmować
  // ich rozróżnianiem:
  class FooL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Wybrano Ble");
    }
```

```
}
class BarL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    t.setText("Wybrano Bla");
  }
}
class BazL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    t.setText("Wybrano Ple");
  }
}
class CMIL implements ItemListener {
  public void itemStateChanged(ItemEvent e) {
    JCheckBoxMenuItem target =
      (JCheckBoxMenuItem)e.getSource();
    String actionCommand = target.getActionCommand();
    if(actionCommand.equals("Chroń"))
      t.setText("Chronić lody! " +
        "Ochrona " + target.getState());
    else if(actionCommand.equals("Ukryj"))
      t.setText("Schować lody! " +
        "Schowane? " + target.getState());
  }
}
public Menus() {
  ML ml = new ML();
  CMIL cmil = new CMIL();
  safety[0].setActionCommand("Chroń");
  safety[0].setMnemonic(KeyEvent.VK_G);
  safety[0].addItemListener(cmil);
  safety[1].setActionCommand("Ukryj");
  safety[1].setMnemonic(KeyEvent.VK_H);
  safety[1].addItemListener(cmil);
  other[0].addActionListener(new FooL());
  other[1].addActionListener(new BarL());
  other[2].addActionListener(new BazL());
  FL fl = new FL();
  int n = 0;
  for(String flavor : flavors) {
    JMenuItem mi = new JMenuItem(flavor);
    mi.addActionListener(fl);
    m.add(mi);
    // Dodanie separatorów:
    if((n++ + 1) % 3 == 0)
      m.addSeparator();
  }
  for(JCheckBoxMenuItem sfty : safety)
    s.add(sfty);
  s.setMnemonic(KeyEvent.VK_A);
  f.add(s);
  f.setMnemonic(KeyEvent.VK_F);
  for(int i = 0; i < file.length; i++) {
    file[i].addActionListener(ml);
    f.add(file[i]);
  }
  mb1.add(f);
  mb1.add(m);
  setJMenuBar(mb1);
```

```
    t.setEditable(false);
    add(t, BorderLayout.CENTER);
    // Przygotowanie do podmiany menu:
    b.addActionListener(new BL());
    b.setMnemonic(KeyEvent.VK_S);
    add(b, BorderLayout.NORTH);
    for(JMenuItem oth : other)
      fooBar.add(oth);
    fooBar.setMnemonic(KeyEvent.VK_B);
    mb2.add(fooBar);
  }
  public static void main(String[] args) {
    run(new Menus(), 300, 200);
  }
} ///:~
```

W tym programie umieściłem wszystkie etykiety menu w tablicach i dodawałem je do menu, przechodząc po każdej tablicy i wywołując metodę `add()` dla każdego elementu `JMenuItem`. Dzięki temu dodawanie i usuwanie elementów staje się trochę mniej żmudne.

Program tworzy nie jeden, ale dwa obiekty paska menu `JMenuBar`, by pokazać, jak można dynamicznie zamieniać paski w trakcie działania programu. Widać tu także wyraźnie, jak pasek menu jest tworzony z obiektów `JMenu`, a każdy z nich powstaje z elementów `JMenuItem`, `JCheckBoxMenuItem` lub nawet kolejnych obiektów `JMenu` (tworzących w ten sposób podmenu). Po złożeniu całego paska `JMenuBar` można go zainstalować w aktualnym programie, wywołując metodę `setJMenuBar()`. Jak widać, w momencie naciśnięcia przycisku wywoływana jest metoda `getJMenuBar()` w celu sprawdzenia, który pasek menu jest aktualnie używany, a następnie w jego miejsce wstawiany jest drugi z pasków.

Zauważ, że przy sprawdzaniu, czy wybranym poleceniem jest *Otwórz*, krytyczna jest poprawna ortografia i wielkość liter, a Java nie zasygnalizuje żadnego błędu, jeśli nie znajdzie się żadne dopasowanie. Tego typu porównywanie etykiet jest częstym źródłem błędów.

Obsługa zaznaczania i odznaczania elementów menu zachodzi automatycznie. W kodzie obsługującym elementy typu `JCheckBoxMenuItem` pokazałem dwa rozwiązania służące do określenia, co zostało zaznaczone: porównywanie etykiet (co nie jest metodą najbezpieczniejszą) i porównanie obiektu docelowego zdarzenia. Do określenia stanu użyć można metody `getState()`. Z kolei aby zmienić stan elementu `JCheckBoxMenuItem`, można wykorzystać metodę `setState()`.

Zdarzenia obsługiwane przez menu są troszkę niespójne i wprowadzają małe zamieszanie: elementy `JMenuItem` korzystają z odbiornika typu `ActionListener`, natomiast elementy `JCheckboxMenuItem` używają odbiorników `ItemListener`. Obiekt `JMenu` również obsługuje odbiornik `ActionListener`, ale przeważnie nic jest to do niczego przydatne. Zwykle przyłącza się odbiorniki do elementów menu typu `JMenuItem`, `JCheckBoxMenuItem` lub `JRadioButtonMenuItem`, ale powyższy przykład pokazuje, jak podłączyć odbiorniki `ItemListener` i `ActionListener` również do innych komponentów menu.

Biblioteka Swing obsługuje skróty klawiaturowe lub „mnemoniki", dzięki którym każdy komponent odziedziczony z `AbstractButton` (przycisk, element menu itp.) można wybrać również za pomocą klawiatury. Stworzenie ich jest bardzo proste: dla elementów

`JMenuItem` można użyć przeciążonego konstruktora przyjmującego jako drugi parametr identyfikator klawisza. Jednakże większość przycisków dziedziczących po `AbstractButton` nie ma tego typu konstruktorów, więc bardziej uniwersalnym podejściem jest wykorzystanie metody `setMnemonic()`. W powyższym przykładzie skróty są zdefiniowane dla przycisku i niektórych elementów menu; automatycznie na tych komponentach pojawiają się znaczniki skrótu.

Wykorzystałem tu również metodę `setActionCommand()`. Wydaje się to trochę dziwne, ponieważ we wszystkich przypadkach to „polecenie" jest dokładnie takie samo, jak etykietka na komponencie menu. Dlaczego po prostu nie użyć etykiet zamiast tych dodatkowych łańcuchów? Problemem jest wielojęzyczność. Przepisując program na inny język, chcielibyśmy, żeby wystarczyło jedynie zmienić etykietkę menu, a nie trzeba było zmieniać niczego w kodzie (co bez wątpienia doprowadziłoby do nowych błędów). Użycie metody `setActionCommand()` ustala jednolitą „akcję polecenia", niezależną od zmian etykiet menu. Cały kod oparty jest na „poleceniach", więc zmiana etykiet menu nic ma na niego żadnego wpływu. Zauważmy, że w powyższym programie nie we wszystkich komponentach sprawdzane są związane z nimi polecenia, więc zestaw poleceń nie jest definiowany.

Większość pracy wykonywana jest przez odbiorniki zdarzeń. Odbiornik `BL` wykonuje zamianę pasków menu. W `ML` przedstawiony jest sposób „Odgadnij, kto woła" przez pobranie źródła zdarzenia `ActionEvent` i zrzutowanie go na `JMenuItem`, a następnie pobranie tekstu polecenia i przepuszczenie go przez serię instrukcji `if`.

Odbiornik `FL` jest bardzo prosty, pomimo że obsługuje wszystkie elementy menu smaków. Takie rozwiązanie jest użyteczne, jeśli logika jest wystarczająco prosta. Jednak generalnie stosuje się rozwiązanie takie jak w `FooL`, `BarL` i `BazL`, które powiązane są wyłącznie z jednym elementem menu, toteż nie potrzeba dodatkowego wykrywania — od razu wiadomo, kto wywołał odbiornik zdarzenia. Nawet przy dużej liczbie wygenerowanych w ten sposób klas ich kod jest przeważnie krótszy i cały program jest bardziej odporny na błędy użytkownika.

Widać już, że kod menu szybko staje się nieczytelny. Jest to kolejny przypadek, kiedy odpowiednim rozwiązaniem jest zastosowanie graficznych narzędzi do budowania interfejsu. Dobre narzędzie powinno także obsługiwać tworzenie menu.

Ćwiczenie 19. Zmień program *Menus.java* tak, aby zamiast pól opcji (ang. *checkbox*) w menu znalazły się przyciski wyboru (ang. *radio button*) (3).

Ćwiczenie 20. Utwórz program, który wczyta zawartość pliku i wyodrębni z niej słowa. Rozprowadź te słowa pomiędzy etykietami w menu i podmenu programu (6).

Menu kontekstowe

Najprostszą metodą zaimplementowania menu kontekstowego (zwanego też „podręcznym") — `JPopupMenu` — jest stworzenie klasy wewnętrznej rozszerzającej `MouseAdapter`, a następnie dodanie obiektu tej klasy do każdego komponentu, który ma posiadać menu podręczne:

```
// : gui/Popup.java
// Tworzenie menu kontekstowego w bibliotece Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Popup extends JFrame {
  private JPopupMenu popup = new JPopupMenu();
  private JTextField t = new JTextField(10);
  public Popup() {
    setLayout(new FlowLayout());
    add(t);
    ActionListener al = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        t.setText(((JMenuItem)e.getSource()).getText());
      }
    };
    JMenuItem m = new JMenuItem("Trele");
    m.addActionListener(al);
    popup.add(m);
    m = new JMenuItem("Morele");
    m.addActionListener(al);
    popup.add(m);
    m = new JMenuItem("Baks");
    m.addActionListener(al);
    popup.add(m);
    popup.addSeparator();
    m = new JMenuItem("Zostań tutaj");
    m.addActionListener(al);
    popup.add(m);
    PopupListener pl = new PopupListener();
    addMouseListener(pl);
    t.addMouseListener(pl);
  }
  class PopupListener extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      maybeShowPopup(e);
    }
    public void mouseReleased(MouseEvent e) {
      maybeShowPopup(e);
    }
    private void maybeShowPopup(MouseEvent e) {
      if(e.isPopupTrigger())
        popup.show(e.getComponent(), e.getX(), e.getY());
    }
  }
  public static void main(String[] args) {
    run(new Popup(), 300, 200);
  }
} ///:~
```

Dla każdego elementu `JMenuItem` ustawiany jest ten sam odbiornik zdarzeń, pobierający tekst z etykiety menu i wstawiający go do pola tekstowego.

Rysowanie

W każdej dobrej bibliotece interfejsu użytkownika rysowanie powinno być prostą czynnością. I tak rzeczywiście jest w bibliotece Swing. Problemem w przykładach rysowania jest to, że obliczenia prowadzące do określenia, gdzie elementy mają się znaleźć, są przeważnie o wiele bardziej złożone niż wywołania procedur rysujących. Poza tym obliczenia są przemieszane z wywołaniami procedur rysujących, toteż może się wydawać, że taki interfejs jest bardziej skomplikowany niż w rzeczywistości.

Dla uproszczenia rozważmy problem przedstawiania danych na ekranie — w naszym przypadku dane te będą dostarczane przez wbudowaną metodę `Math.sin()`, która odpowiada matematycznej funkcji sinus. Aby całość była troszkę bardziej interesująca i żeby jeszcze raz zademonstrować, jak łatwo używa się komponentów Swing, na spodzie formatki umieszczony będzie suwak pozwalający dynamicznie określić, ile cykli sinusoidy ma być wyświetlanych. Dodatkowo po zmianie okna sinusoida będzie dopasowywać do niego swój rozmiar.

Mimo że rysować można na każdym komponencie dziedziczącym po `JComponent`, to jeśli chcemy mieć zwykłą powierzchnię do rysowania, przeważnie będziemy wywodzić ją z `JPanel`. Jedyną metodą, którą trzeba przesłonić, jest `paintComponent()` wywoływana za każdym razem, kiedy komponent musi być przerysowany (ale normalnie nic trzeba się tym zajmować, ponieważ jest to obsługiwane przez Swing). Przy wywołaniu Swing przekazuje do niej obiekt `Graphics`, którego można następnie używać do rysowania bądź malowania na dostępnej powierzchni.

W poniższym przykładzie cała logika związana z malowaniem znajduje się w klasie `SineDraw`. Natomiast klasa `SineWave` konfiguruje program i kontrolkę suwaka. Wewnątrz `SineDraw` metoda `setCycles()` stanowi punkt zaczepienia dla innego obiektu — w tym przypadku kontrolki suwaka — wykorzystywanego do sterowania liczbą cykli.

```
//: gui/SineWave.java
// Rysowanie w Swingu, zastosowanie suwaka JSlider.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

class SineDraw extends JPanel {
  private static final int SCALEFACTOR = 200;
  private int cycles;
  private int points;
  private double[] sines;
  private int[] pts;
  public SineDraw() { setCycles(5); }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    int maxWidth = getWidth();
    double hstep = (double)maxWidth / (double)points;
    int maxHeight = getHeight();
    pts = new int[points];
    for(int i = 0; i < points; i++)
      pts[i] =
        (int)(sines[i] * maxHeight/2 * .95 + maxHeight/2);
```

```
      g.setColor(Color.RED);
      for(int i = 1; i < points; i++) {
        int x1 = (int)((i - 1) * hstep);
        int x2 = (int)(i * hstep);
        int y1 = pts[i-1];
        int y2 = pts[i];
        g.drawLine(x1, y1, x2, y2);
      }
    }
    public void setCycles(int newCycles) {
      cycles = newCycles;
      points = SCALEFACTOR * cycles * 2;
      sines = new double[points];
      for(int i = 0; i < points; i++) {
        double radians = (Math.PI / SCALEFACTOR) * i;
        sines[i] = Math.sin(radians);
      }
      repaint();
    }
  }

  public class SineWave extends JFrame {
    private SineDraw sines = new SineDraw();
    private JSlider adjustCycles = new JSlider(1, 30, 5);
    public SineWave() {
      add(sines);
      adjustCycles.addChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent e) {
          sines.setCycles(
            ((JSlider)e.getSource()).getValue());
        }
      });
      add(BorderLayout.SOUTH, adjustCycles);
    }
    public static void main(String[] args) {
      run(new SineWave(), 700, 400);
    }
  } ///:~
```

Wszystkie zmienne składowe i tablice używane są przy obliczeniach punktów sinusoidy: `cycles` określa żądaną liczbę pełnych cykli, `points` stanowi całkowitą liczbę punktów do narysowania, `sines` zawiera wartości funkcji sinus, `pts` zawiera współrzędne *y* punktów, które zostaną narysowane na panelu. Metoda `setCycles()` tworzy obie tablice na podstawie liczby potrzebnych punktów i wypełnia wartościami tablicę `sines`. Przez wywołanie metody `repaint()`, `setCycles()` wymusza wywołanie metody `paintComponent()`, a więc wykonana zostanie reszta obliczeń i komponent zostanie przerysowany na nowo.

Pierwszą rzeczą, którą trzeba wykonać przy przesłonięciu `paintComponent()`, jest wywołanie bazowej wersji tej metody. Potem możemy robić, co nam się tylko podoba. Przeważnie oznacza to użycie metod obiektu `Graphics`, które odnaleźć można w dokumentacji dla klasy `java.awt.Graphics` (w dokumentacji JDK dostępnej z *http://java.sun.com*), do rysowania i malowania pikseli na panelu `JPanel`. Widać tutaj, że niemal cały kod związany jest z wykonaniem obliczeń; jedyne dwie metody, które rzeczywiście operują na ekranie, to `setColor()` i `drawLine()`. Przypomina to tworzenie własnego programu wyświetlającego dane graficzne — większość czasu poświęca się na wyliczenie tego, co właściwie chce się narysować, a rzeczywisty proces rysowania okazuje się bardzo prosty.

Tworząc ten przykładowy program, większość czasu poświęciłem na uzyskanie wyświetlenia sinusoidy. Kiedy już udało mi się to osiągnąć, pomyślałem, że dobrze byłoby umożliwić dynamiczną zmianę liczby cykli. Moje doświadczenia związane z robieniem podobnych rzeczy w innych językach programowania sprawiły, że podszedłem do tego trochę niechętnie, ale ostatecznie okazało się, że była to najłatwiejsza część projektu. Stworzyłem jedynie suwak `JSlider` (parametry konstruktora to odpowiednio najmniejsza, największa i początkowa wartość suwaka, ale ma on również inne konstruktory) i umieściłem go na formatce. Następnie zauważyłem w dokumentacji JDK, że jedynym odbiornikiem zdarzeń jest `ChangeListener`, informowany za każdym razem, kiedy suwak jest przesuwany na tyle, żeby zmienił wartość. Jedyna metoda obsługi nazywa się oczywiście `stateChanged()`. Uzyskuje ona obiekt zdarzenia `ChangeEvent`, przez co można odszukać źródło zdarzenia i pobrać nową wartość. Nowa wartość przekazywana jest do wywołania metody `setCycles()` obiektu `sines` i następuje odrysowywanie panelu.

Przeważnie okazuje się, że większość problemów graficznych można rozwiązać w podobny sposób i ogólnie jest on bardzo prosty, nawet jeśli trzeba do tego wykorzystać komponenty, których nigdy wcześniej się nie używało.

Jeśli zadanie jest bardziej złożone, to dostępne są też bardziej wyszukane rozwiązania, wliczając w to dodatkowe komponenty JavaBeans od innych producentów oraz bibliotekę Java 2D. Rozwiązania te wykraczają jednak poza zakres niniejszej książki, ale należy się nimi zainteresować, jeśli pisanie kodu rysującego staje się zbyt uciążliwe.

Ćwiczenie 21. Zmodyfikuj *SineWave.java* tak, by zmienić klasę `SineDraw` w kontrolkę JavaBean, dodając metody `getter` i `setter` (5).

Ćwiczenie 22. Utwórz aplikację, używając klasy `SwingConsole`. Umieść tam trzy suwaki dla trzech składowych RGB (kolory czerwony, niebieski i zielony). Pozostała część formatki niech będzie zajęta przez panel `JPanel`, który wyświetla kolor określony przez trzy suwaki. Umieść również nieedytowalne pola tekstowe, pokazujące aktualne wartości składowych RGB (7).

Ćwiczenie 23. Bazując na programie *SineWave.java*, napisz aplikację, która wyświetli na ekranie obracający się kwadrat. Jeden suwak programu powinien regulować szybkość obrotu, a drugi — rozmiar kwadratu (8).

Ćwiczenie 24. Na pewno nieobca Ci jest zabawka do rysowania z dwoma pokrętłami sterującymi pionowym i poziomym ruchem rysika. Stwórz podobną zabawkę, bazując na programie *SineWave.java*. Zamiast pokręteł użyj suwaków. Dołącz przycisk wymazujący cały rysunek (7).

Ćwiczenie 25. Bazując na programie *SineWave.java*, stwórz program (aplikację wykorzystującą klasę `SwingConsole`), który będzie wyświetlać sinusoidę sprawiającą wrażenie, jak gdyby przesuwała się poza okno programu (tworząc efekt podobny do oscyloskopu). Działanie animacji oprzyj na klasie `java.util.Timer`. Szybkość animacji należy regulować przy użyciu suwaka (komponentu `java.swing.JSlider`) (8).

Ćwiczenie 26. Zmodyfikuj program z poprzedniego ćwiczenia w taki sposób, aby w oknie aplikacji mogło być tworzonych wiele paneli prezentujących animacje. Program powinien pozwalać regulować liczbę wyświetlanych paneli z poziomu wiersza poleceń (5).

Ćwiczenie 27. Zmodyfikuj program z ćwiczenia 25. w taki sposób, aby animacją zarządzał obiekt `javax.swing.Timer`. Zwróć uwagę na różnice pomiędzy nim a obiektem `java.util.Timer` (5).

Ćwiczenie 28. Utwórz klasę kości (samą klasę, bez interfejsu). Utwórz w programie pięć kości i rzucaj je kolejno. Wyrysuj na ekranie krzywą reprezentującą sumę oczek w kolejnych rzutach i pokaż zmiany krzywej w miarę wzrostu liczby rzutów (7).

Okna dialogowe

Okno dialogowe jest takim rodzajem okna, które „wyskakuje" z innego. Może służyć do obsługi jakiegoś specyficznego zagadnienia, nie przyczyniając się do zaśmiecania oryginalnego okna jego szczegółami. Okna dialogowe są powszechnie używane w środowiskach okienkowych.

Aby stworzyć okno dialogowe, należy je odziedziczyć z klasy `JDialog`, która jest po prostu innym rodzajem okna, podobnie jak `JFrame`. Obiekt klasy `JDialog` posiada menedżera ułożenia (domyślnie jest to `BorderLayout`) i można do niego dodawać odbiorniki zdarzeń. Oto bardzo prosty przykład:

```
// : gui/Dialogs.java
// Tworzenie i stosowanie okien dialogowych.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

class MyDialog extends JDialog {
  public MyDialog(JFrame parent) {
    super(parent, "Moje okno dialogowe", true);
    setLayout(new FlowLayout());
    add(new JLabel("Oto moje okienko dialogowe"));
    JButton ok = new JButton("OK");
    ok.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        dispose(); // Zamyka okno dialogowe
      }
    });
    add(ok);
    setSize(150,125);
  }
}

public class Dialogs extends JFrame {
  private JButton b1 = new JButton("Okno dialogowe");
  private MyDialog dlg = new MyDialog(null);
  public Dialogs() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        dlg.setVisible(true);
      }
    });
    add(b1);
  }
```

```
  public static void main(String[] args) {
    run(new Dialogs(), 125, 75);
  }
} ///:~
```

Po stworzeniu obiektu `JDialog` trzeba wywołać metodę `setVisible(true)`, żeby wyświetlić i aktywować okno. Przy zamykaniu okna musi ono zwolnić zajmowane zasoby, do czego służy wywołanie `dispose()`.

Poniższy przykład jest bardziej złożony. Okno dialogowe składa się z siatki (opartej na `GridLayout`) zbudowanej ze specjalnego rodzaju przycisków zdefiniowanych tutaj jako klasa `ToeButton`. Przyciski te same rysują wokół siebie ramkę, przy tym w zależności od stanu są albo puste, albo mają na środku `x` lub `o`. Początkowo są puste i w zależności od tego, czyj jest ruch, zmieniają się na `x` lub `o`. Jednak po kliknięciu przycisku `x` zmieni się w `o` i odwrotnie (dzięki temu gra w kółko i krzyżyk jest tylko odrobinę bardziej irytująca). Dodatkowo okno dialogowe może zostać skonfigurowane na dowolną liczbę kolumn i wierszy, zmieniając liczby w głównym oknie aplikacji.

```
// : gui/TicTacToe.java
// Okna dialogowe z własnymi komponentami.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class TicTacToe extends JFrame {
  private JTextField
    rows = new JTextField("3"),
    cols = new JTextField("3");
  private enum State { BLANK, XX, OO }
  static class ToeDialog extends JDialog {
    private State turn = State.XX; // Zaczynamy od kolejki 'x'-ów
    ToeDialog(int cellsWide, int cellsHigh) {
      setTitle("Gra");
      setLayout(new GridLayout(cellsWide, cellsHigh));
      for(int i = 0; i < cellsWide * cellsHigh; i++)
        add(new ToeButton());
      setSize(cellsWide * 50, cellsHigh * 50);
      setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    }
    class ToeButton extends JPanel {
      private State state = State.BLANK;
      public ToeButton() { addMouseListener(new ML()); }
      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int
          x1 = 0, y1 = 0,
          x2 = getSize().width - 1,
          y2 = getSize().height - 1;
        g.drawRect(x1, y1, x2, y2);
        x1 = x2/4;
        y1 = y2/4;
        int wide = x2/2, high = y2/2;
        if(state == State.XX) {
          g.drawLine(x1, y1, x1 + wide, y1 + high);
          g.drawLine(x1, y1 + high, x1 + wide, y1);
```

```
        }
        if(state == State.OO)
          g.drawOval(xl, yl, xl + wide/2, yl + high/2);
      }
      class ML extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
          if(state == State.BLANK) {
            state = turn;
            turn =
              (turn == State.XX ? State.OO : State.XX);
          }
          else
            state =
              (state == State.XX ? State.OO : State.XX);
          repaint();
        }
      }
    }
    class BL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        JDialog d = new ToeDialog(
          new Integer(rows.getText()),
          new Integer(cols.getText()));
        d.setVisible(true);
      }
    }
    public TicTacToe() {
      JPanel p = new JPanel();
      p.setLayout(new GridLayout(2,2));
      p.add(new JLabel("Wiersze", JLabel.CENTER));
      p.add(rows);
      p.add(new JLabel("Kolumny", JLabel.CENTER));
      p.add(cols);
      add(p, BorderLayout.NORTH);
      JButton b = new JButton("dalej");
      b.addActionListener(new BL());
      add(b, BorderLayout.SOUTH);
    }
    public static void main(String[] args) {
      run(new TicTacToe(), 200, 200);
    }
  } ///:~
```

Ponieważ składowe statyczne mogą być jedynie na zewnętrznym poziomie klasy, to klasy zagnieżdżone nie mogą posiadać statycznych zmiennych ani statycznych klas zagnieżdżonych.

Metoda `paintComponent()` rysuje kwadrat wokół panelu oraz znak x bądź o. Pełno jest tu żmudnych obliczeń, ale całość jest bardzo prosta.

Naciśnięcie przycisku myszy jest przechwytywane przez odbiornik `MouseListener`, który najpierw sprawdza, czy panel ma już coś na sobie narysowane. Jeśli nie, to do okna nadrzędnego kierowane jest zapytanie o to, czyja jest teraz kolej, i ta informacja jest używana do ustalenia stanu przycisku `ToeButton`. Poprzez mechanizm klas wewnętrznych

przycisk może następnie sięgnąć do swojej klasy otaczającej i zmienić turę. Jeśli przycisk wyświetla już znak x lub o, to zostanie on zamieniany. W obliczeniach widać przydatność trójargumentowej instrukcji `if-else` opisanej w rozdziale „Operatory". Po zmianie stanu przycisk jest odrysowywany na nowo.

Konstruktor obiektu typu `ToeDialog` jest bardzo prosty: dodaje do `GridLayout` tyle przycisków, ile zażądamy, i rozszerza okno w każdą stronę o 50 pikseli dla każdego przycisku.

Klasa `TicTacToe` konfiguruje całą aplikację, tworząc pola tekstowe (do wprowadzania liczby kolumn i wierszy siatki przycisków) oraz przycisk *start* z odbiornikiem zdarzeń. Kiedy przycisk zostanie wciśnięty, dane z pól tekstowych są pobierane, a ponieważ mają postać łańcuchów, to zamieniane są na liczby przy użyciu konstruktora klasy `Integer` w wersji przyjmującej argument typu `String`.

Okna dialogowe plików

Niektóre systemy operacyjne mają kilka specjalnych, wbudowanych okien dialogowych przeznaczonych do obsługi wyboru takich rzeczy, jak: czcionka, kolor, drukarki itp. Jednakże potencjalnie wszystkie graficzne systemy operacyjne obsługują otwieranie i zapisywanie plików, toteż klasa Javy `JFileChooser` grupuje je, ułatwiając ich używanie.

Poniższa aplikacja wykorzystuje dwie postacie okna dialogowego `JFileChooser`. Jedno służy do otwierania, a drugie do zapisywania plików. Większość kodu powinna być już zrozumiała, a wszystkie ciekawe rzeczy znajdują się w odbiornikach zdarzeń dwóch przycisków:

```
//: gui/FileChooserTest.java
// Działanie okien dialogowych wyboru plików.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class FileChooserTest extends JFrame {
  private JTextField
    fileName = new JTextField(),
    dir = new JTextField();
  private JButton
    open = new JButton("Otwórz"),
    save = new JButton("Zapisz");
  public FileChooserTest() {
    JPanel p = new JPanel();
    open.addActionListener(new OpenL());
    p.add(open);
    save.addActionListener(new SaveL());
    p.add(save);
    add(p, BorderLayout.SOUTH);
    dir.setEditable(false);
    fileName.setEditable(false);
    p = new JPanel();
    p.setLayout(new GridLayout(2,1));
    p.add(fileName);
    p.add(dir);
    add(p, BorderLayout.NORTH);
```

```
    }
    class OpenL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        JFileChooser c = new JFileChooser();
        // Okno dialogowe otwierania pliku:
        int rVal = c.showOpenDialog(FileChooserTest.this);
        if(rVal == JFileChooser.APPROVE_OPTION) {
          fileName.setText(c.getSelectedFile().getName());
          dir.setText(c.getCurrentDirectory().toString());
        }
        if(rVal == JFileChooser.CANCEL_OPTION) {
          fileName.setText("Nacisnąłeś Cancel");
          dir.setText("");
        }
      }
    }
    class SaveL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        JFileChooser c = new JFileChooser();
        // Okno dialogowe zapisu pliku:
        int rVal = c.showSaveDialog(FileChooserTest.this);
        if(rVal == JFileChooser.APPROVE_OPTION) {
          fileName.setText(c.getSelectedFile().getName());
          dir.setText(c.getCurrentDirectory().toString());
        }
        if(rVal == JFileChooser.CANCEL_OPTION) {
          fileName.setText("Nacisnąłeś Cancel");
          dir.setText("");
        }
      }
    }
    public static void main(String[] args) {
      run(new FileChooserTest(), 250, 150);
    }
  } ///:~
```

Okno dialogowe `JFileChooser` może być używane na wiele różnych sposobów, wliczając w to użycie filtrów w celu zawężenia zestawu dopuszczalnych nazw plików.

Aby uzyskać okno dialogowe typu *otwórz plik*, wywołuje się metodę `showOpenDialog()`, z kolei by stworzyć okno typu *zapisz plik*, należy użyć `showSaveDialog()`. Polecenia te nie zwracają sterowania, dopóki okno dialogowe nie zostanie zamknięte. Metody `getSelectedFile()` i `getCurrentDirectory()` zapewniają dwa sposoby na to, żeby dowiedzieć się, jaki był wynik operacji. Jeśli zwrócą `null`, oznacza to, że użytkownik anulował wybór z okna.

Ćwiczenie 29. W dokumentacji JDK dla `javax.swing` odszukaj komponent `JColorChooser`. Napisz program z przyciskiem, który otwiera okno dialogowe pozwalające na wybranie koloru (3).

HTML w komponentach Swing

Każdy komponent, który może pobierać tekst, przyjmuje również tekst HTML. Zostanie on sformatowany zgodnie z zasadami HTML. Oznacza to, że bardzo łatwo można nadać komponentowi Swing wymyślny tekst. Przykładowo:

```
//: gui/HTMLButton.java
// Tekst HTML na komponentach Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class HTMLButton extends JFrame {
  private JButton b = new JButton(
    "<html><b><font size=+2>" +
    "<center>Ahoj!<br><i>Naciśnij mnie!");
  public HTMLButton() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        add(new JLabel("<html>" +
          "<i><font size=+4>Łubudu!"));
        // Wymuszenie ponownego ułożenia komponentów
        // z uwzględnieniem nowej etykiety:
        validate();
      }
    });
    setLayout(new FlowLayout());
    add(b);
  }
  public static void main(String[] args) {
    run(new HTMLButton(), 200, 500);
  }
} ///:~
```

Tekst należy rozpocząć od `<html>` i można już dalej używać normalnych znaczników HTML. Przy tym nie ma obowiązku wstawiania normalnych znaczników zamykających.

Odbiornik zdarzenia `ActionListener` umieszcza na formatce nową etykietę `JLabel`, która również zawiera tekst HTML. Jednak, ponieważ nie jest ona dodawana w trakcie konstrukcji, trzeba wywołać metodę `validate()`, aby wymusić ponowne rozmieszczenie komponentów (a w rezultacie wyświetlenie nowej etykiety).

Składni HTML można używać również w komponentach: `JTabbedPane`, `JMenuItem`, `JToolTip`, `JRadioButton` i `JCheckBox`.

Ćwiczenie 30. Napisz program, który zademonstruje zastosowanie tekstu HTML na wszystkich komponentach wymienionych w poprzednim akapicie (3).

Suwaki i wskaźniki postępu

Suwak (którego używaliśmy już w przykładzie z sinusoidą) pozwala użytkownikowi na wprowadzanie danych przez przesuwanie wskaźnika w dwóch kierunkach, co w niektórych sytuacjach jest dosyć intuicyjne (na przykład przy sterowaniu głośnością). Pasek

postępu wyświetla natomiast informacje, używając metafory „pusty-pełny", więc użytkownik może obejrzeć stan. Moim ulubionym przykładem jest połączenie suwaka ze wskaźnikiem postępu tak, żeby wskaźnik postępu zmieniał się odpowiednio do przesunięcia suwaka. Poniższy przykład przy okazji demonstruje możliwości okna dialogowego monitora postępu `ProgressMonitor`:

```
//: gui/Progress.java
// Suwaki, wskaźniki postępu i monitory postępu.
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Progress extends JFrame {
  private JProgressBar pb = new JProgressBar();
  private ProgressMonitor pm = new ProgressMonitor(
    this, "Monitorowanie postępu", "Test", 0, 100);
  private JSlider sb =
    new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
  public Progress() {
    setLayout(new GridLayout(2,1));
    add(pb);
    pm.setProgress(0);
    pm.setMillisToPopup(1000);
    sb.setValue(0);
    sb.setPaintTicks(true);
    sb.setMajorTickSpacing(20);
    sb.setMinorTickSpacing(5);
    sb.setBorder(new TitledBorder("Przesuń mnie"));
    pb.setModel(sb.getModel()); // Wspólny model
    add(sb);
    sb.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        pm.setProgress(sb.getValue());
      }
    });
  }
  public static void main(String[] args) {
    run(new Progress(), 300, 200);
  }
} ///:~
```

Kluczem do złączenia obydwu komponentów jest współdzielenie ich modelu w wierszu:

```
pb.setModel(sb.getModel());
```

Oczywiście można by sterować obydwoma przez odbiornik zdarzeń, ale w tak prostej sytuacji użycie wspólnego modelu jest dużo łatwiejsze. `ProgressMonitor` nie posiada modelu i w jego przypadku trzeba uciec się do odbiornika. Zauważ, że monitor `ProgressMonitor` przesuwa się jedynie wprzód, a po osiągnięciu końca wskaźnika okno monitora się zamyka.

Komponent `JProgressBar` jest bardzo prosty, natomiast `JSlider` posiada wiele różnych opcji, takich jak orientacja oraz znaczniki podziałki (główne i poboczne). Zauważ przy okazji, jak łatwo można dodać ramkę z tytułem.

Ćwiczenie 31. Utwórz „asymptotyczny wskaźnik postępu", który przesuwa się coraz wolniej, w miarę jak zbliża się do końca. Dodaj losowe zaburzenia tak, by czasami wydawało się, że zaczyna przyspieszać (8).

Ćwiczenie 32. Zmodyfikuj program *Progress.java* tak, aby nie wykorzystywał dzielenia modelu, ale zamiast tego używał odbiornika zdarzeń do połączenia suwaka i wskaźnika postępu (6).

Zmiana stylu interfejsu

„Przełączany wygląd aplikacji" (ang. *pluggable look & feel*) pozwala na emulowanie we własnym programie wyglądu i zachowania różnych systemów operacyjnych. Możliwa jest nawet dynamiczna zmiana wyglądu w trakcie pracy programu. Jednakże przeważnie chcemy zrobić jedną z dwóch rzeczy: albo wybrać styl jednolity dla wszystkich platform (tzn. charakterystyczny dla Swinga metaliczny wygląd), albo wybrać wygląd właściwy systemowi, w którym program jest uruchomiony, tak aby przypominał normalne programy tego systemu (bez wątpienia jest to najlepsze rozwiązanie, gdyż odpowiada ono oczekiwaniom użytkowników programu). Kod pozwalający wybrać któryś z tych stylów jest bardzo prosty. Jednak należy go wykonać *przed* stworzeniem jakichkolwiek kontrolek, ponieważ kontrolki są tworzone na podstawie aktualnie wybranego stylu i nie zostaną zmienione tylko dlatego, że zachciało nam się zmienić styl wyglądu w czasie pracy programu (taki przypadek jest bardziej skomplikowany i rzadko spotykany, został opisany w książkach poświęconych bibliotece Swing).

Jeśli chcemy używać wspólnego dla wszystkich platform metalicznego stylu charakterystycznego dla programów Swing, to właściwie nic nie musimy robić — jest on przyjmowany domyślnie. Jeśli natomiast chcemy użyć stylu odpowiadającego aktualnemu środowisku pracy[8], wystarczy umieścić w programie poniższy kod, przeważnie na początku metody `main()`, ale przed dodaniem jakichkolwiek kontrolek:

```
try {
  UIManager.setLookAndFeel(UIManager.
    getSystemLookAndFeelClassName());
} catch(Exception e) {
  throw new RuntimeException(e);
}
```

Nie ma potrzeby umieszczania czegokolwiek w sekcji `catch`, ponieważ jeśli próba wybrania innego stylu nie powiedzie się, `UIManager` przyjmie styl domyślny. Jednak w trakcie testowania programu może się przydać obsługa tego wyjątku, więc warto wstawić tam chociaż polecenie wyświetlenia komunikatu o błędzie.

W poniższym programie można wybrać styl, podając odpowiedni parametr w wierszu poleceń. Program pokazuje, jak wyglądają niektóre kontrolki w wybranych stylach.

```
//: gui/LookAndFeel.java
// Wybieranie różnych stylów wizualnych.
// {Args: motif}
import javax.swing.*;
```

[8] Nie zawsze naśladownictwo stylu charakterystycznego dla danej platformy jest udane.

```
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class LookAndFeel extends JFrame {
  private String[] choices =
    "ene due rike fake dero baks".split(" ");
  private Component[] samples = {
    new JButton("JButton"),
    new JTextField("JTextField"),
    new JLabel("JLabel"),
    new JCheckBox("JCheckBox"),
    new JRadioButton("Radio"),
    new JComboBox(choices),
    new JList(choices),
  };
  public LookAndFeel() {
    super("Style");
    setLayout(new FlowLayout());
    for(Component component : samples)
      add(component);
  }
  private static void usageError() {
    System.out.println(
      "Stosowanie:LookAndFeel [cross|system|motif]");
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length == 0) usageError();
    if(args[0].equals("cross")) {
      try {
        UIManager.setLookAndFeel(UIManager.
          getCrossPlatformLookAndFeelClassName());
      } catch(Exception e) {
        e.printStackTrace();
      }
    } else if(args[0].equals("system")) {
      try {
        UIManager.setLookAndFeel(UIManager.
          getSystemLookAndFeelClassName());
      } catch(Exception e) {
        e.printStackTrace();
      }
    } else if(args[0].equals("motif")) {
      try {
        UIManager.setLookAndFeel("com.sun.java."+
          "swing.plaf.motif.MotifLookAndFeel");
      } catch(Exception e) {
        e.printStackTrace();
      }
    } else usageError();
    // Zauważ, że styl trzeba ustawić przed
    // utworzeniem jakiegokolwiek komponentu.
    run(new LookAndFeel(), 300, 300);
  }
} ///:~
```

Jednym z rozwiązań jest bezpośrednie podanie nazwy stylu, tak jak powyżej dla `MotifLookAndFeel`. Jednak tylko ten styl oraz domyślny styl metaliczny mogą być używane legalnie na dowolnej platformie. Mimo że nazwy stylów dla środowisk Windows i Macintosh są zdefiniowane, mogą one być używane tylko w danym środowisku (są one zwracane przez metodę `getSystemLookAndFeelClassName()`, kiedy program jest uruchamiany na danej platformie).

Można również stworzyć własny pakiet stylów, na przykład tworząc zestaw bibliotek dla firmy, która chce, by wygląd jej produktów wyróżniał je od innych programów. Wymaga to sporego nakładu pracy i znacznie wykracza poza zakres tej książki (co więcej, wykracza to poza zakres większości książek poświęconych Swingowi!).

Drzewka, tabele i schowek

Krótkie wprowadzenie i prezentacje komponentów odpowiadających tym elementom interfejsu znajduje się w suplementach towarzyszących książce, a publikowanych na stronie *www.MindView.net*.

JNLP oraz Java Web Start

Programista ma możliwość cyfrowego podpisywania apletów (dla bezpieczeństwa); jest to omawiane w suplemencie dla tego rozdziału publikowanym w witrynie *www.MindView.net*. Podpisane cyfrowo aplety dają ogromne możliwości i w efektywny sposób mogą zastępować zwyczajne aplikacje, niemniej jednak muszą być wykonywane w przeglądarkach WWW, co powoduje dodatkowy narzut związany z koniecznością uruchamiania przeglądarki, a jednocześnie ogranicza interfejs użytkownika apletu, sprawiając, że może być on trudny do obsługi. Przeglądarka WWW ma grupę własnych pasków menu i pasków narzędziowych, które będą widoczne nad apletem[9].

Protokół JNLP (ang. *Java Network Lanuch Protocol* — protokół uruchamiania sieciowego aplikacji Javy) rozwiązuje ten problem, zachowując jednocześnie korzyści, jakie dają aplety. Tworząc aplikacje JNLP, można pobrać i zainstalować niezależną aplikację na komputerze użytkownika. Po zainstalowaniu aplikację można będzie uruchamiać z wiersza poleceń, po kliknięciu ikony umieszczonej na pulpicie lub za pośrednictwem menedżera aplikacji instalowanego wraz z implementacją JNLP. Istnieje nawet możliwość uruchomienia aplikacji z witryny, z której została pobrana.

Aplikacje JNLP mogą dynamicznie — podczas pracy — pobierać zasoby z internetu, a jeśli użytkownik ma połączenie z siecią, to automatycznie może być sprawdzany numer wersji programu. Oznacza to, że aplikacje JNLP posiadają wszelkie zalety zarówno apletów, jak i niezależnych aplikacji.

System klienta musi podchodzić do aplikacji JNLP równie ostrożnie jak do apletów. Z tego względu podczas ich wykonywania stosowany jest ten sam system zabezpieczeń co w przypadku apletów. Podobnie jak aplety, aplikacje JNLP mogą być udostępniane w formie

[9] Ten podrozdział opracował Jeremy Meyer.

podpisanych cyfrowo plików JAR, dzięki czemu użytkownik może mieć możliwość zaufania ich twórcy. Niemniej jednak, w odróżnieniu od apletów, aplikacje tego typu mogą prosić o uzyskanie pozwolenia dostępu do zasobów lokalnych, nawet jeśli nie zostaną podpisane; jest to możliwe dzięki wykorzystaniu usług oferowanych przez interfejs programistyczny JNLP. Tyle że użytkownik musi wyrazić zgodę na operacje podczas działania programu).

JNLP to opis protokołu, a nie jego implementacja, dlatego też, by z niego korzystać, należy zdobyć implementację tego protokołu. Java Web Start (w skrócie JAWS) jest oficjalną implementacją wzorcową, rozprowadzaną jako część Javy SE5. Jeśli używasz jej do tworzenia aplikacji, powinieneś zadbać, by pliki *JAR* znalazły się w katalogu podanym w zmiennej CLASSPATH; najprościej dodać do ścieżki CLASSPATH położenie *javaws.jar* względem standardowej instalacji w podkatalogu *jre/lib*. Udostępniając aplikację JNLP na serwerze WWW, należy upewnić się, że serwer obsługuje typ MIME `application/x-java-jnlp-file`. Jeśli używanym serwerem jest najnowsza wersja Tomcata (*http://jakarta.apache.org/tomcat/*), to obsługa tego typu MIME będzie domyślnie skonfigurowana. Warto zajrzeć do dokumentacji używanego serwera.

Tworzenie aplikacji JNLP nie jest trudne. Należy w tym celu stworzyć zwyczajną aplikację, zapisać ją w pliku JAR, a następnie dodać plik uruchomieniowy — prosty plik XML udostępniający wszystkie informacje konieczne do pobrania i zainstalowania aplikacji. Jeśli plik JAR nie będzie podpisany, to próbując uzyskać dostęp do każdego z typów zasobów lokalnego komputera, trzeba posługiwać się odpowiednimi usługami JNLP API.

Poniżej przedstawiłem kolejną wersję przykładu *FileChooserTest.java*, która otwiera okno dialogowe za pośrednictwem usługi JNLP, dzięki czemu klasę można udostępniać jako aplikację JNLP umieszczoną w niepodpisanym pliku JAR.

```
// : gui/jnlp/JnlpFileChooser.java
// Otwieranie plików na lokalnym komputerze klienta za
// pośrednictwem JNLP.
// {Requires: javax.jnlp.FileOpenService;
// W obrębie ścieżki CLASSPATH musi znajdować się pakiet javaws.jar}
// Aby utworzyć plik jnlpfilechooser.jar, wykonaj polecenia:
// cd ..
// cd ..
// jar cvf gui/jnlp/jnlpfilechooser.jar gui/jnlp/*.class
package gui.jnlp;
import javax.jnlp.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class JnlpFileChooser extends JFrame {
  private JTextField fileName = new JTextField();
  private JButton
    open = new JButton("Otwórz"),
    save = new JButton("Zapisz");
  private JEditorPane ep = new JEditorPane();
  private JScrollPane jsp = new JScrollPane();
  private FileContents fileContents;
  public JnlpFileChooser() {
```

```
    JPanel p = new JPanel();
    open.addActionListener(new OpenL());
    p.add(open);
    save.addActionListener(new SaveL());
    p.add(save);
    jsp.getViewport().add(ep);
    add(jsp, BorderLayout.CENTER);
    add(p, BorderLayout.SOUTH);
    fileName.setEditable(false);
    p = new JPanel();
    p.setLayout(new GridLayout(2,1));
    p.add(fileName);
    add(p, BorderLayout.NORTH);
    ep.setContentType("text");
    save.setEnabled(false);
  }
  class OpenL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      FileOpenService fs = null;
      try {
        fs = (FileOpenService)ServiceManager.lookup(
          "javax.jnlp.FileOpenService");
      } catch(UnavailableServiceException use) {
        throw new RuntimeException(use);
      }
      if(fs != null) {
        try {
          fileContents = fs.openFileDialog(".",
            new String[]{"txt", "*"});
          if(fileContents == null)
            return;
          fileName.setText(fileContents.getName());
          ep.read(fileContents.getInputStream(), null);
        } catch(Exception exc) {
          throw new RuntimeException(exc);
        }
        save.setEnabled(true);
      }
    }
  }
  class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      FileSaveService fs = null;
      try {
        fs = (FileSaveService)ServiceManager.lookup(
          "javax.jnlp.FileSaveService");
      } catch(UnavailableServiceException use) {
        throw new RuntimeException(use);
      }
      if(fs != null) {
        try {
          fileContents = fs.saveFileDialog(".",
            new String[]{"txt"},
            new ByteArrayInputStream(
              ep.getText().getBytes()),
            fileContents.getName());
          if(fileContents == null)
            return;
```

```
            fileName.setText(fileContents.getName());
          } catch(Exception exc) {
            throw new RuntimeException(exc);
          }
        }
      }
    }
    public static void main(String[] args) {
      JnlpFileChooser fc = new JnlpFileChooser();
      fc.setSize(400, 300);
      fc.setVisible(true);
    }
  } ///:~
```

Należy zwrócić uwagę, że klasy `FileOpenService` oraz `FileCloseService` są importowane z pakietu `javax.jnlp`, a w kodzie programu nie ma ani jednego jawnego odwołania do okna dialogowego `JFileChooser`. O możliwość użycia obu usług wykorzystywanych w programie należy poprosić za pośrednictwem metody `ServiceManager.lookup()`, a dostęp do zasobów lokalnego komputera uzyskiwany jest wyłącznie za pośrednictwem obiektów zwróconych przez tę metodę. W naszym przypadku zapis i odczyt plików lokalnych odbywa się za pośrednictwem interfejsu `FileContent` udostępnianego przez JNLP. Każda próba bezpośredniego dostępu do pliku, dajmy na to przy użyciu obiektu `File` bądź `FileReader`, skończyłaby się tak samo, jak próba użycia tych klas w niepodpisanym aplecie — zgłoszeniem wyjątku `SecurityException`. Aby mieć możliwość korzystania z tych klas bez konieczności posługiwania się usługami JNLP, należy podpisać plik JAR.

Oznaczone jako komentarz w kodzie źródłowym *JnlpFileChooser.java* polecenie `jar` utworzy potrzebny plik JAR. Oto kod takiego pliku, odpowiedni dla aplikacji z poprzedniego przykładu:

```
//:! gui/jnlp/filechooser.jnlp
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec = "1.0+"
  codebase="file:C:/AAA-TIJ4/code/gui/jnlp"
  href="filechooser.jnlp">
  <information>
    <title>FileChooser demo application</title>
    <vendor>Mindview Inc.</vendor>
    <description>
      Jnlp File chooser Application
    </description>
    <description kind="short">
      Demonstrates opening, reading and writing a text file
    </description>
    <icon href="mindview.gif"/>
    <offline-allowed/>
  </information>
  <resources>
    <j2se version="1.3+"
      href="http://java.sun.com/products/autodl/j2se"/>
    <jar href="jnlpfilechooser.jar" download="eager"/>
  </resources>
  <application-desc
    main-class="gui.jnlp.JnlpFileChooser"/>
</jnlp>
//:~
```

Powyższy plik znajdziesz w pakiecie kodu źródłowego rozprowadzanego wraz z niniejszą książką w pliku *filechooser.jnlp* (jest tam zapisany bez pierwszego i ostatniego wiersza), w tym samym katalogu co plik JAR. Jak widać, jest to plik XML zawierający jeden element `<jnlp>`. Zawiera kilka elementów zagnieżdżonych, których znaczenie, w większości przypadków, nie wymaga żadnych dodatkowych wyjaśnień.

Atrybut `spec` elementu `jnlp` informuje, z jaką wersją protokołu JNLP aplikacja może współpracować. Atrybut `codebase` wskazuje lokalizację (URL) zawierającą plik uruchomieniowy oraz wszelkie pozostałe zasoby. W naszym przypadku określa on nazwę katalogu na lokalnym komputerze; łatwo poznać po tym aplikację w fazie testów. *Pamiętaj, że powinieneś zmienić ciąg określający katalog na taki, który będzie zgodny z układem katalogów w Twoim systemie, inaczej programu nie uda się załadować.* Atrybut `href` musi zawierać nazwę danego pliku uruchomieniowego.

Element `information` zawiera wiele podelementów podających informacje dotyczące aplikacji. Są one używane przez konsolę administracyjną Java Web Start (lub analogiczne narzędzie), która instaluje aplikację JNLP i daje użytkownikom możliwość uruchamiania jej z wiersza poleceń za pomocą skrótu itp.

Element `resources` spełnia podobną funkcję co znacznik `applet` stosowany w dokumentach HTML. Element `j2se` określa wersję JDK konieczną do uruchomienia aplikacji, a element `jar` podaje nazwę pliku JAR, w którym została umieszczona klasa aplikacji. Element `jar` zawiera także atrybut `download`. Może on przyjmować dwie wartości — `eager` (ładowanie „wczesne") lub `lazy` (ładowanie „opóźnione") — informujące, czy przed uruchomieniem aplikacji konieczne jest pobranie całego pliku JAR.

Atrybut `application-desc` informuje, która klasa jest klasą wykonywalną bądź „punktem wejścia" pliku JAR.

Kolejnym przydatnym podelementem, który można umieścić wewnątrz elementu `jnlp`, jest `security` (nie został on przedstawiony na powyższym przykładzie). Oto postać tego znacznika:

```
<security>
   <all-permissions/>
</security>
```

Znacznik ten jest stosowany w sytuacjach, gdy aplikacja jest umieszczona w podpisanym pliku JAR. W przypadku naszej przykładowej aplikacji nie jest on potrzebny, gdyż dostęp do zasobów lokalnych odbywa się za pośrednictwem usług JNLP.

W plikach uruchomieniowych aplikacji JNLP stosowane są także inne elementy. Szczegółowe informacje na ich temat można znaleźć na stronie *http://java.sun.com/products/javawebstart/download-spec.html*.

Do uruchomienia programu potrzebna jest jeszcze strona WWW z odnośnikiem hipertekstowym do pliku *.jnlp*. Może ona wyglądać tak (znów bez pierwszego i ostatniego wiersza):

```
//:! gui/jnlp/filechooser.html
<html>
Aby zbudować pakiet jnlpfilechooser.jar, wykonaj instrukcje zapisane
w pliku JnlpFileChooser.java, a następnie:
```

```
<a href="filechooser.jnlp">kliknij tu</a>
</html>
//:~
```

Po pobraniu aplikacji można ją skonfigurować przy wykorzystaniu konsoli administracyjnej. W przypadku korzystania z Java Web Start w systemie Windows podczas drugiego uruchamiania aplikacji zostaniesz zapytany, czy należy utworzyć na pulpicie skrót do niej. Istnieje możliwość zmiany tego zachowania.

W tej części rozdziału przedstawione zostały jedynie dwie usługi JNLP, w aktualnej implementacji protokołu jest ich siedem. Każda z nich służy do obsługi konkretnego zadania, takiego jak drukowanie czy też kopiowanie i wklejanie danych ze schowka. Po dodatkowe informacje odsyłam do witryny *http://java.sun.com*.

Swing a współbieżność

Programowanie z użyciem biblioteki Swing oznacza zawsze korzystanie z wątków. Zaznaczyłem to już na początku rozdziału, zalecając przekazywanie wszelkich zadań do wątku dyspozytora zdarzeń Swing za pomocą metody `SwingUtilities.invokeLater()`. Jednakże fakt, że obiekty wątków (`Thread`) nie są używane jawnie, może czasami doprowadzić sytuacji, w której zagadnienia związane z wielowątkowością nagle nas zaskoczą. Trzeba więc zawsze pamiętać, że Swing uruchamia wątek rozprowadzający zdarzenia, który działa zawsze i obsługuje zdarzenia generowane przez komponenty biblioteki. Mając świadomość obecności takiego wątku łatwiej unikniemy ryzyka sytuacji hazardowych i zakleszczeń.

Zajmiemy się teraz właśnie zagadnieniami związanymi z wielowątkowością aplikacji korzystających z biblioteki Swing.

Zadania długotrwałe

Jednym z powszechniejszych błędów programowania graficznego interfejsu użytkownika jest wykorzystywanie wątku dyspozytora zadań do nieopatrznego uruchomienia długotrwałego zadania. Oto prosty przykład takiej wpadki:

```
//: gui/LongRunningTask.java
// Niewłaściwie zaprojektowany program.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingConsole.*;

public class LongRunningTask extends JFrame {
  private JButton
    b1 = new JButton("Uruchom długotrwałe zadanie"),
    b2 = new JButton("Zakończ długotrwałe zadanie");
  public LongRunningTask() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
```

```
          TimeUnit.SECONDS.sleep(3);
        } catch(InterruptedException e) {
          System.out.println("Zadanie przerwane");
          return;
        }
        System.out.println("Zadanie zakończone");
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        // Przerwać, czy nie przerwać?
        Thread.currentThread().interrupt();
      }
    });
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
  }
  public static void main(String[] args) {
    run(new LongRunningTask(), 200, 150);
  }
} ///:~
```

Po naciśnięciu przycisku b1 wątek dyspozytora zdarzeń zostanie zaangażowany do wykonywania długotrwałego zadania. Efekt widać od razu: przycisk nawet nie „odskakuje", bo wątek rozprowadzający zdarzenia, który normalnie zająłby się odrysowaniem przycisku, jest zajęty. Nie można nic zrobić w programie, w szczególności nie można nacisnąć przycisku b2, bo program nie zareaguje nijak aż do czasu zakończenia zadania, w które zaangażował się dyspozytor zdarzeń. Próba rozwiązania problemu przez przerwanie zadania dyspozytora zdarzeń w wyniku obsługi przycisku b2 jest oczywiście nieskuteczna.

Właściwym rozwiązaniem jest naturalnie uruchomienie długotrwałych zadań w osobnych wątkach. Poniżej mamy przykład wykorzystania wykonawcy pojedynczego wątku, który automatycznie kolejkuje zadania oczekujące na wykonanie i kolejno je realizuje:

```
// gui/InterruptableLongRunningTask.java
// Długotrwałe zadania w oddzielnych wątkach.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingConsole.*;

class Task implements Runnable {
  private static int counter = 0;
  private final int id = counter++;
  public void run() {
    System.out.println(this + " uruchomione");
    try {
      TimeUnit.SECONDS.sleep(3);
    } catch(InterruptedException e) {
      System.out.println(this + " przerwane");
      return;
    }
    System.out.println(this + " zakończone");
  }
```

```
  public String toString() { return "Zadanie " + id; }
  public long id() { return id; }
};

public class InterruptableLongRunningTask extends JFrame {
  private JButton
    b1 = new JButton("Uruchom długotrwałe zadanie"),
    b2 = new JButton("Zakończ długotrwałe zadanie");
  ExecutorService executor =
    Executors.newSingleThreadExecutor();
  public InterruptableLongRunningTask() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Task task = new Task();
        executor.execute(task);
        System.out.println(task + " dodane do kolejki");
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        executor.shutdownNow(); // ręcznie
      }
    });
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
  }
  public static void main(String[] args) {
    run(new InterruptableLongRunningTask(), 220, 150);
  }
} ///:~
```

To znacznie lepszy układ, ale po naciśnięciu przycisku b2 dochodzi do wywołania shutdownNow() na rzecz obiektu wykonawcy ExecutorService, co blokuje tego wykonawcę. Gdybyśmy spróbowali przekazać do niego kolejne zadania, spowodowalibyśmy wyjątek. Naciśnięcie przycisku b2 skutecznie blokuje możliwość kontynuowania programu i odwołuje wszystkie zadania oczekujące na realizację. Potrzebujemy tu ni mniej, ni więcej tylko interfejsu Callable i mechanizmu Future, opisywanego w rozdziale „Współbieżność". Definiujemy więc nową klasę TaskManager, zawierającą *krotki* (ang. *tuples*) przechowujące implementacje Callable reprezentujące zadania i skojarzone z nimi obiekty Future reprezentujące wyniki realizacji tych zadań. Zastosowanie krotek pozwala na powiązanie wyników z pierwotnymi zadaniami, co umożliwia pozyskanie dodatkowych informacji o zadaniu, niedostępnych w obiektach Future. Oto przykład:

```
//: net/mindview/util/TaskItem.java
// Krotka wiążąca wynik zadania (Future) z samym zadaniem (Callable).
package net.mindview.util;
import java.util.concurrent.*;

public class TaskItem<R,C extends Callable<R>> {
  public final Future<R> future;
  public final C task;
  public TaskItem(Future<R> future, C task) {
    this.future = future;
    this.task = task;
  }
} ///:~
```

W bibliotece `java.util.concurrent` obiekt oryginalnego zadania nie jest podtrzymywany po wykonaniu, kiedy powstaje już gotowy obiekt `Future`. Aby wymusić podtrzymanie pierwotnego obiektu zadania, zachowujemy jego referencję w krotce.

Klasa `TaskManager` została umieszczona w bibliotece `net.mindview.util`, przez co jest dostępna jako narzędzie ogólnego przeznaczenia:

```
//: net/mindview/util/TaskManager.java
// Zarządzanie kolejką zadań.
package net.mindview.util;
import java.util.concurrent.*;
import java.util.*;

public class TaskManager<R,C extends Callable<R>>
extends ArrayList<TaskItem<R,C>> {
  private ExecutorService exec =
    Executors.newSingleThreadExecutor();
  public void add(C task) {
    add(new TaskItem<R,C>(exec.submit(task),task));
  }
  public List<R> getResults() {
    Iterator<TaskItem<R,C>> items = iterator();
    List<R> results = new ArrayList<R>();
    while(items.hasNext()) {
      TaskItem<R,C> item = items.next();
      if(item.future.isDone()) {
        try {
          results.add(item.future.get());
        } catch(Exception e) {
          throw new RuntimeException(e);
        }
        items.remove();
      }
    }
    return results;
  }
  public List<String> purge() {
    Iterator<TaskItem<R,C>> items = iterator();
    List<String> results = new ArrayList<String>();
    while(items.hasNext()) {
      TaskItem<R,C> item = items.next();
      // Pozostawienie zakończonych zadań
      // dla potrzeb raportowania wyników:
      if(!item.future.isDone()) {
        results.add("Odwoływanie zadania " + item.task);
        item.future.cancel(true); // Może zostać przerwane
        items.remove();
      }
    }
    return results;
  }
} ///:~
```

Sam `TaskManager` to kontener `ArrayList` krotek `TaskItem`. Obiekt menedżera zawiera też pojedynczy egzemplarz wykonawcy (`Executor`), więc wywołanie na rzecz obiektu `TaskManager` metody `add()` z obiektem zadania (`Callable`) powoduje przekazanie tego zadania

do kolejki wykonawcy i zachowanie wyniku Future wraz z pierwotnym obiektem zadania we wspólnej krotce. Dzięki temu w przyszłości można się odwoływać do obiektu zrealizowanego zadania. Przykładem może być metoda purge() wykorzystująca metodę toString() obiektu zadania.

Całość możemy wykorzystać do zarządzania długotrwałymi zadaniami z naszego przykładu:

```
//: gui/InterruptableLongRunningCallable.java
// Interfejs Callables w służbie długotrwałych zadań.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

class CallableTask extends Task
implements Callable<String> {
  public String call() {
    run();
    return "Wartość zwracana " + this;
  }
}

public class
InterruptableLongRunningCallable extends JFrame {
  private JButton
    b1 = new JButton("Uruchom długotrwałe zadanie"),
    b2 = new JButton("Zakończ długotrwałe zadanie"),
    b3 = new JButton("Pobierz wyniki");
  private TaskManager<String,CallableTask> manager =
    new TaskManager<String,CallableTask>();
  public InterruptableLongRunningCallable() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        CallableTask task = new CallableTask();
        manager.add(task);
        System.out.println(task + " dodane do kolejki");
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(String result : manager.purge())
          System.out.println(result);
      }
    });
    b3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // Próbne wywołanie metody Task:
        for(TaskItem<String,CallableTask> tt :
            manager)
          tt.task.id(); // Rzutowanie niepotrzebne
        for(String result : manager.getResults())
          System.out.println(result);
      }
```

```
    });
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
    add(b3);
  }
  public static void main(String[] args) {
    run(new InterruptableLongRunningCallable(), 220, 150);
  }
} ///:~
```

Jak widać, obiekt `CallableTask` to to samo co obiekt `Task`, z tym że zwraca wynik — w naszym przypadku jest to ciąg znaków identyfikujący zadanie.

Do rozwiązania podobnych problemów powołano do życia narzędzia `SwingWorkers` (niewchodzące w skład standardowej biblioteki Swing, ale możliwe do pobrania z witryny WWW firmy Sun) i *Foxtrot* (zobacz *http://foxtrot.sourceforge.net*), ale w czasie przygotowywania niniejszej książki narzędzia te nie były jeszcze przystosowane do wykorzystywania mechanizmu `Callable-Future` z Javy SE5.

Niejednokrotnie wypadałoby dać użytkownikowi wskazówkę, rodzaj sygnalizacji co do działania zadania i informacje o jego postępie. Normalnie służy do tego wskaźnik postępu `JProgressBar`, ewentualnie okienko `ProgressMonitor`. W poniższym przykładzie użyjemy tego ostatniego:

```
// : gui/MonitoredLongRunningCallable.java
// Wyświetlanie wskaźnika postępu długotrwałego zadania.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

class MonitoredCallable implements Callable<String> {
  private static int counter = 0;
  private final int id = counter++;
  private final ProgressMonitor monitor;
  private final static int MAX = 8;
  public MonitoredCallable(ProgressMonitor monitor) {
    this.monitor = monitor;
    monitor.setNote(toString());
    monitor.setMaximum(MAX - 1);
    monitor.setMillisToPopup(500);
  }
  public String call() {
    System.out.println(this + " uruchomione");
    try {
      for(int i = 0; i < MAX; i++) {
        TimeUnit.MILLISECONDS.sleep(500);
        if(monitor.isCanceled())
          Thread.currentThread().interrupt();
        final int progress = i;
        SwingUtilities.invokeLater(
          new Runnable() {
```

```
            public void run() {
              monitor.setProgress(progress);
            }
          }
        );
      }
    } catch(InterruptedException e) {
      monitor.close();
      System.out.println(this + " przerwane");
      return "Wynik: " + this + " przerwane";
    }
    System.out.println(this + " zakończone");
    return "Wynik: " + this + " zakończone";
  }
  public String toString() { return "Zadanie " + id; }
};

public class MonitoredLongRunningCallable extends JFrame {
  private JButton
    b1 = new JButton("Uruchom długotrwałe zadanie"),
    b2 = new JButton("Zakończ długotrwałe zadanie"),
    b3 = new JButton("pobierz wyniki");
  private TaskManager<String,MonitoredCallable> manager =
    new TaskManager<String,MonitoredCallable>();
  public MonitoredLongRunningCallable() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        MonitoredCallable task = new MonitoredCallable(
          new ProgressMonitor(
            MonitoredLongRunningCallable.this,
            "Długotrwałe zadanie", "", 0, 0)
        );
        manager.add(task);
        System.out.println(task + " dodane do kolejki");
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(String result : manager.purge())
          System.out.println(result);
      }
    });
    b3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        for(String result : manager.getResults())
          System.out.println(result);
      }
    });
    setLayout(new FlowLayout());
    add(b1);
    add(b2);
    add(b3);
  }
  public static void main(String[] args) {
    run(new MonitoredLongRunningCallable(), 220, 500);
  }
} ///:~
```

Konstruktor `MonitoredCallable` przyjmuje za pośrednictwem argumentu wywołania obiekt klasy `ProgressMonitor`, a metoda `call()` co pół sekundy aktualizuje stan monitora postępu. Zauważ, że obiekt klasy `MonitoredCallable` to osobne zadanie i jako takie nie powinno próbować sterować elementami interfejsu użytkownika wprost, stąd w operacji przekazania nowych informacji o stanie do monitora obecność wywołania `SwingUtilities.invokeLater()`. Przewodnik po bibliotece Swing, publikowany przez firmę Sun (*Swing Tutorial*, szukaj w witrynie *http://java.sun.com*), prezentuje podejście alternatywne, polegające na użyciu komponentu Swing `Timer`, który okresowo sprawdza stan zadania i aktualizuje monitor.

Gdyby w okienku monitora naciśnięty został przycisk *Cancel*, metoda `monitor.isCancelled()` zwróciłaby wartość logiczną `true`. W naszym przykładzie reakcją na to jest zwyczajne przerwanie zadania wywołaniem metody `interrupt()` na rzecz bieżącego wątku. Program wyląduje wtedy w stosownej w klauzuli `catch`, gdzie monitor jest zamykany wywołaniem `close()`.

Reszta kodu w zasadzie się nie zmieniła, tyle że teraz tworzenie obiektu `ProgressMonitor` odbywa się w ramach konstruktora klasy `MonitoredLongRunningCallable`.

Ćwiczenie 33. Zmodyfikuj program *InterruptableLongRunningCallable.java* tak, aby uruchamiał wszystkie zadania nie kolejno, a współbieżnie (6).

Wizualizacja wielowątkowości interfejsu użytkownika

W poniższym przykładzie implementujemy interfejs `Runnable` w klasie dziedziczącej po `JPanel`, aby uzyskać klasę umożliwiającą wyświetlanie różnych kolorów na powierzchni komponentu. Aplikacja pobiera argumenty z wiersza poleceń i na ich podstawie określa rozmiar siatki kolorów oraz czas oczekiwania (realizowanego za pomocą metody `sleep()`) pomiędzy kolejnymi zmianami kolorów. Modyfikując obie te wartości, można odkryć niektóre dziwne i, być może, niewytłumaczalne cechy wątków:

```
//: gui/ColorBoxes.java
// Wizualizacja wielowątkowości.
// {Args: 12 50}
import javax.swing.*;
import java.awt.*;
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

class CBox extends JPanel implements Runnable {
  private int pause;
  private static Random rand = new Random();
  private Color color = new Color(0);
  public void paintComponent(Graphics g) {
    g.setColor(color);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  public CBox(int pause) { this.pause = pause; }
  public void run() {
    try {
```

```
        while(!Thread.interrupted()) {
          color = new Color(rand.nextInt(0xFFFFFF));
          repaint(); // Asynchroniczne żądanie odrysowania komponentu
          TimeUnit.MILLISECONDS.sleep(pause);
        }
      } catch(InterruptedException e) {
        // Dopuszczalny sposób zakończenia
      }
    }
  }

  public class ColorBoxes extends JFrame {
    private int grid = 12;
    private int pause = 50;
    private static ExecutorService exec =
      Executors.newCachedThreadPool();
    public ColorBoxes() {
      setLayout(new GridLayout(grid, grid));
      for(int i = 0; i < grid * grid; i++) {
        CBox cb = new CBox(pause);
        add(cb);
        exec.execute(cb);
      }
    }
    public static void main(String[] args) {
      ColorBoxes boxes = new ColorBoxes();
      if(args.length > 0)
        boxes.grid = new Integer(args[0]);
      if(args.length > 1)
        boxes.pause = new Integer(args[1]);
      run(boxes, 500, 400);
    }
  } ///:~
```

Program `ColorBoxes` konfiguruje menedżer układu `GridLayout` tak, aby utworzył siatkę komórek o równej liczbie w pionie i poziomie. Następnie wypełnia siatkę odpowiednią liczbą obiektów `CBox`, przekazując do każdego z nich wartość `pause`. W metodzie `main()` można zauważyć, że zmienne `pause` oraz `grid` mają wartości domyślne, które można zmienić, przekazując własne argumenty wiersza poleceń.

Wszystkie czynności są wykonywane w obiektach `CBox`. Klasa ta dziedziczy po `JPanel` i implementuje interfejs `Runnable`, dzięki czemu każdy obiekt `JPanel` jest jednocześnie niezależnym zadaniem. Zadania te są uruchamiane za pośrednictwem wykonawcy z pulą wątków (`ExecutorService`).

Kolor bieżącej komórki to `cColor`. Kolory są tworzone wywołaniami konstruktora klasy `Color`, do którego przekazuje się 24-bitową liczbę — u nas jest ona dobierana losowo.

Metoda `paintComponent()` jest całkiem prosta — określa ona kolor na podstawie `cColor`, a następnie wypełnia nim całą powierzchnię komponentu.

W metodzie `run()` została zdefiniowana nieskończona pętla, która przypisuje nową, losową wartość składowej `cColor`, a następnie wyświetla wybrany kolor, wywołując metodę `repaint()`. Po odświeżeniu komponentu działanie wątku jest wstrzymywane (poprzez wywołanie metody `sleep()`) na czas podany w wierszu wywołania programu.

Na szczególną uwagę zasługuje wywołanie metody `repaint()` w obrębie metody `run()`. Na pierwszy rzut oka dochodzi tu do tworzenia całego mnóstwa wątków, z których każdy wymusza odrysowanie swojego kawałka siatki. Naruszałoby to regułę powstrzymywania się od bezpośredniego ingerowania w stan komponentów interfejsu i korzystania z pośrednictwa kolejki zdarzeń, ale owe wątki wcale nie modyfikują zasobu wspólnego. Wywołanie `repaint()` nie wymusza natychmiastowego odrysowania danego obszaru, a jedynie oznacza komponent, którego wartość będzie rozpatrywana przy okazji najbliższej obsługi zdarzenia odrysowania okna. Dlatego program nie powoduje żadnych problemów z wielowątkowością w obrębie Swing.

Kiedy dyspozytor zdarzeń podejmie właściwe odrysowanie (`paint()`), w pierwszej kolejności wywoła metodę `paintComponent()`, a potem `paintBorder()` i `paintChildren()`. Jeśli w komponencie pochodnym zachodzi potrzeba przesłonięcia metody `paint()`, trzeba pamiętać o wywołaniu w wersji przesłoniętej wersji z klasy bazowej — tak aby zagwarantować wykonanie właściwej procedury odrysowania.

Dzięki wykorzystaniu elastycznego projektu i połączenia wątków z komponentami `JPanel` można eksperymentować, tworząc dowolną liczbę wątków. (W rzeczywistości jest pewne ograniczenie liczby wątków, które mogą być obsługiwane przez wirtualną maszynę Javy i nie będą wpływać na efektywność jej działania.)

Powyższy program jest jednocześnie interesującym testem wydajności, gdyż w zależności od implementacji wielowątkowości zastosowanych w różnych wirtualnych maszynach Javy (i na różnych platformach), różnice w jego efektywności i zachowaniu mogą być bardzo wyraźnie widoczne.

Ćwiczenie 34. Zmodyfikuj program *ColorBoxes.java* tak, aby na początku rozsypał w obszarze okna pewną liczbę punktów („gwiazdek"), a następnie zmieniał kolory tych „gwiazdek" (4).

Programowanie wizualne i komponenty JavaBean

Chyba udało mi się już przekonać Cię, jak bardzo Java jest przydatna przy tworzeniu kodu nadającego się do wielokrotnego wykorzystania. Klasa jest jednostką kodu, która najlepiej się do tego nadaje. Jest ona spójną całością złożoną z charakterystyk (pól) oraz zachowań (metod), której można powtórnie użyć zarówno bezpośrednio przez kompozycję, jak i przez dziedziczenie.

Dziedziczenie i polimorfizm są zasadniczymi elementami programowania obiektowego, jednak w większości przypadków przy tworzeniu aplikacji tak naprawdę zależy nam na *komponentach*, które robią dokładnie to, co chcemy. Chcemy je składać w projekcie tak samo, jak inżynier elektronik składa ze sobą układy na płytce elektronicznej. Wydaje się również, że powinien istnieć jakiś sposób, aby usprawnić taki „montażowy" styl programowania.

„Programowanie wizualne" stało się popularne, *bardzo* popularne, dzięki językowi Visual Basic (VB) firmy Microsoft i powstałemu zaraz po nim środowisku drugiej generacji Borland Delphi (główne źródło inspiracji dla projektu JavaBean). W takich narzędziach komponenty są reprezentowane graficznie. Jest to rozsądne rozwiązanie, gdyż komponenty przeważnie wyświetlają jakiś element graficzny, jak przycisk lub pole tekstowe. Najczęściej reprezentacja graficzna komponentu jest dokładnie taka sama, jak jego wygląd w działającym programie. Zatem częścią procesu programowania wizualnego jest przeciąganie komponentu z palety na formatkę. Narzędzie tworzenia aplikacji (zintegrowane środowisko programistyczne) samo już wtedy dopisze za nas odpowiedni kod, odpowiedzialny za stworzenie tego komponentu w programie.

Przeciągnięcie komponentu na formatkę zazwyczaj nie wystarcza do stworzenia całego programu. Często trzeba też zmienić charakterystyki komponentu, takie jak jego kolor, napis, jaki ma na sobie, ustawienia bazy danych, do której jest podłączony itp. Charakterystyki, które można modyfikować w trakcie projektowania, nazywane są *właściwościami* (ang. *properties*). Zintegrowane środowiska programistyczne pozwalają na wygodną modyfikację właściwości komponentu i dbają o to, by zapisać całą jego konfigurację w programie tak, że może ona zostać odtworzona przy uruchamianiu.

W tym momencie jasne staje się już, że obiekt to coś więcej niż tylko charakterystyka — jest to także zbiór zachowań. W trakcie projektowania zachowania komponentów graficznych są częściowo reprezentowane przez *zdarzenia* (ang. *events*) oznaczające, że „oto coś, co może się stać z komponentem". Zazwyczaj programista decyduje, co ma nastąpić, kiedy wystąpi dane zdarzenie, wiążąc z tym zdarzeniem własny kod.

A oto kluczowy element: zintegrowane środowiska programistyczne używają mechanizmu refleksji, aby dynamicznie „przesłuchać" komponent i dowiedzieć się, jakie atrybuty i zdarzenia posiada. Kiedy takie narzędzie uzyska już niezbędne informacje na temat atrybutów i zdarzeń komponentu, może wyświetlić właściwości i pozwolić nam je modyfikować (zapisując ich stan przed kompilacją programu); podobnie potrafi wyświetlić zdarzenia. Przeważnie polega to na dwukrotnym kliknięciu nazwy zdarzenia, w wyniku czego narzędzie tworzy dla nas ogólne ramy kodu, który będzie powiązany z danym zdarzeniem. Trzeba jeszcze tylko napisać kod wykonywany w chwili wystąpienia zdarzenia.

Dzięki temu dużą część naszej pracy mogą wykonać narzędzia programistyczne. W rezultacie możemy się skoncentrować na tym, jak ma wyglądać sam program i co ma robić, pozostawiając narzędziu zajęcie się szczegółami połączenia tego w całość. Powodem, dla którego wizualne środowiska programistyczne są tak popularne, jest to, że bardzo przyspieszają proces tworzenia aplikacji, a z pewnością tworzenie interfejsu użytkownika, chociaż często również innych części aplikacji.

Czym jest komponent JavaBean?

„Kiedy kurz opadnie", okazuje się, że tak naprawdę komponent jest tylko blokiem kodu, przeważnie w postaci klasy. Kluczowym zagadnieniem jest możliwość odkrycia przez narzędzie do tworzenia aplikacji, jakie ten komponent posiada właściwości i zdarzenia. Aby stworzyć komponent VB, programista musiał napisać dość skomplikowany fragment kodu, uwzględniając określone konwencje, aby udostępnić właściwości i zdarzenia (w miarę upływu lat było to coraz prostsze). Delphi było narzędziem wizualnego pro-

gramowania drugiej generacji, w którym język był stworzony z myślą o wizualnym programowaniu, toteż o wiele łatwiej tworzyć w nim wizualne komponenty. Jednakże dopiero Java wraz z JavaBeans wprowadziła tworzenie wizualnych komponentów na wyższy poziom, ponieważ komponent JavaBean jest najzwyczajniejszą klasą. Nie ma potrzeby pisania dodatkowego kodu lub konieczności użycia specjalnych rozszerzeń języka, aby coś stało się komponentem JavaBean. Jedyną rzeczą, jaką rzeczywiście należy zrobić, jest lekkie skorygowanie sposobu nazywania metod. To właśnie nazwa metody mówi narzędziu, czy jest to właściwość, zdarzenie czy też po prostu zwyczajna metoda.

W dokumentacji JDK powyższa konwencja nazewnicza jest błędnie określana jako „wzorzec projektowy". Jest to dość niefortunne sformułowanie, ponieważ i bez tego wzorce projektowe są wystarczająco trudnym wyzwaniem (zobacz *Thinking in Patterns with Java*, do ściągnięcia pod adresem *www.MindView.net*). Nie jest to jednak żaden wzorzec projektowy, a tylko prosta konwencja nazewnicza:

1. Dla właściwości o nazwie xxx tworzy się przeważnie dwie metody: `getXxx()` oraz `setXxx()`. Zatem nazwę właściwości można uzyskać z członu po „get" lub „set", zamieniając pierwszą literę na małą. Typ zwracany przez metodę „get" jest taki sam, jak typ argumentu metody „set". Nazwa właściwości oraz typ dla metod „get" i „set" nie są ze sobą powiązane.
2. Dla właściwości typu `boolean` można tak jak powyżej używać prefiksów „get" i „set", ale można też zamiast „get" użyć „is".
3. Zwyczajne metody komponentu nie stosują się do powyższej konwencji, ale są publiczne.
4. Dla zdarzeń stosowane jest podejście typowe dla odbiorników zdarzeń w bibliotece Swing. Jest to dokładnie to samo, co już widzieliśmy: aby obsłużyć zdarzenie `FooBarEvent`, użyj `addFooBarListener(FooBarListener)` oraz `removeFooBarListener(FooBarListener)`. W większości przypadków wystarczą nam wbudowane zdarzenia i ich odbiorniki, ale można też stworzyć własne i zdefiniować dla nich interfejsy odbiornika.

Kierując się powyższymi wskazaniami, możemy stworzyć prosty komponent:

```
//: frogbean/Frog.java
// Prosty komponent JavaBean.
package frogbean;
import java.awt.*;
import java.awt.event.*;

class Spots {}

public class Frog {
  private int jumps;
  private Color color;
  private Spots spots;
  private boolean jmpr;
  public int getJumps() { return jumps; }
  public void setJumps(int newJumps) {
    jumps = newJumps;
  }
  public Color getColor() { return color; }
  public void setColor(Color newColor) {
```

```
    color = newColor;
  }
  public Spots getSpots() { return spots; }
  public void setSpots(Spots newSpots) {
    spots = newSpots;
  }
  public boolean isJumper() { return jmpr; }
  public void setJumper(boolean j) { jmpr = j; }
  public void addActionListener(ActionListener l) {
    // ...
  }
  public void removeActionListener(ActionListener l) {
    // ...
  }
  public void addKeyListener(KeyListener l) {
    // ...
  }
  public void removeKeyListener(KeyListener l) {
    // ...
  }
  // "Zwyczajna" metoda publiczna:
  public void croak() {
    System.out.println("Kum!");
  }
} ///:~
```

Przede wszystkim widać, że jest to zwyczajna klasa. Przeważnie wszystkie pola będą prywatne i dostępne wyłącznie poprzez metody. Podążając za konwencją nazewniczą, właściwościami tej klasy są: `jumps`, `color`, `spots` oraz `jumper` (zauważ zmianę wielkości pierwszej litery nazwy właściwości). Pomimo iż w pierwszych trzech przypadkach nazwa właściwości jest taka sama, jak nazwa wewnętrznego identyfikatora, to w przypadku `jumper` widać, że nazwa właściwości nie wymusza użycia żadnego określonego identyfikatora dla zmiennej wewnętrznej (a nawet nie wymusza tego, żeby dla danej właściwości *była* zdefiniowana jakakolwiek zmienna wewnętrzna).

Patrząc na nazwy metod dla odbiorników zdarzeń „add" i „remove", zauważamy, że zdarzeniami obsługiwanymi przez ten komponent są `ActionEvent` oraz `KeyEvent`. Na koniec widać, że zwykła metoda `croak()` również jest częścią komponentu. Dzieje się tak dlatego, że jest to metoda publiczna, a nie dlatego, że jest zgodna z jakimkolwiek schematem nazewnictwa.

Wydobycie informacji o komponencie poprzez klasę Introspector

Najbardziej krytycznym momentem w przypadku komponentów JavaBeans jest chwila przeciągnięcia komponentu z palety i upuszczenia go na formatkę. Zintegrowane środowisko programistyczne musi mieć możliwość stworzenia komponentu (może to zrobić, jeśli posiada on domyślny konstruktor), a następnie, bez odwoływania się do jego kodu źródłowego, musi wydobyć wszelkie informacje potrzebne do stworzenia arkusza właściwości i obsługiwanych zdarzeń.

Część rozwiązania wydaje się oczywista, jeśli pamiętamy rozdział „Informacje o typach": mechanizm *refleksji* pozwala odkrywać wszelkie metody anonimowej klasy. Jest to idealne rozwiązanie problemu komponentów JavaBeans, niewymagające użycia żadnych dodatkowych słów kluczowych, które są wymagane w innych wizualnych językach programowania. W rzeczywistości głównym powodem, dla którego refleksja została dodana do Javy, była właśnie obsługa komponentów (chociaż obsługuje również serializację obiektów i zdalne wywołania metod, jest też przydatna w zwykłych zadaniach programistycznych). Można się więc spodziewać, że twórca zintegrowanego środowiska programistycznego musi używać tego mechanizmu wobec każdego komponentu JavaBean i odszukiwać jego metody, aby odnaleźć właściwości i zdarzenia takiego komponentu.

Jest to oczywiście wykonalne, ale projektanci chcieli dostarczyć standardowe narzędzie nie tylko po to, żeby sprawić, by komponenty JavaBean były łatwiejsze w użyciu, ale również żeby dostarczyć standardowy mechanizm do tworzenia bardziej złożonych komponentów. Narzędziem tym jest klasa `Introspector`, a jej najważniejszą metodą jest statyczna metoda `getBeanInfo()`. Można jej przekazać odwołanie do klasy, a wtedy klasa ta zostanie z góry na dół przeszukana i w wyniku dostaniemy obiekt `BeanInfo`, w którym znajdziemy wszystkie odnalezione właściwości, metody i zdarzenia.

Przeważnie nie musimy się tym wcale zajmować — w przypadku większości gotowych komponentów nie musimy wcale wiedzieć, co się z nimi dzieje. Po prostu przeciągamy je na formularz, następnie konfigurujemy ich właściwości i piszemy metody obsługi zdarzeń, które nas interesują. Jednakże użycie klasy `Introspector` do wyświetlenia informacji o kontrolce jest pouczającym i ciekawym doświadczeniem. Oto narzędzie, które to robi:

```
//: gui/BeanDumper.java
// Introspekcja komponentu JavaBean.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.lang.reflect.*;
import static net.mindview.util.SwingConsole.*;

public class BeanDumper extends JFrame {
  private JTextField query = new JTextField(20);
  private JTextArea results = new JTextArea();
  public void print(String s) { results.append(s + "\n"); }
  public void dump(Class<?> bean) {
    results.setText("");
    BeanInfo bi = null;
    try {
      bi = Introspector.getBeanInfo(bean, Object.class);
    } catch(IntrospectionException e) {
      print("Nie można podejrzeć komponentu " +  bean.getName());
      return;
    }
    for(PropertyDescriptor d: bi.getPropertyDescriptors()){
      Class<?> p = d.getPropertyType();
      if(p == null) continue;
      print("Typ właściwości:\n  " + p.getName() +
        "\nNazwa właściwości:\n  " + d.getName());
      Method readMethod = d.getReadMethod();
```

```
        if(readMethod != null)
          print("Metoda odczytująca:\n  " + readMethod);
        Method writeMethod = d.getWriteMethod();
        if(writeMethod != null)
          print("Metoda ustawiająca:\n  " + writeMethod);
        print("====================");
      }
      print("Metody publiczne:");
      for(MethodDescriptor m : bi.getMethodDescriptors())
        print(m.getMethod().toString());
      print("====================");
      print("Obsługa zdarzeń:");
      for(EventSetDescriptor e: bi.getEventSetDescriptors()){
        print("Typ odbiornika:\n  " +
          e.getListenerType().getName());
        for(Method lm : e.getListenerMethods())
          print("Metoda odbiornika:\n  " + lm.getName());
        for(MethodDescriptor lmd :
            e.getListenerMethodDescriptors() )
          print("Deskryptor metody:\n  " + lmd.getMethod());
        Method addListener= e.getAddListenerMethod();
        print("Metoda dodająca odbiornik:\n  " + addListener);
        Method removeListener = e.getRemoveListenerMethod();
        print("Metoda usuwająca odbiornik:\n  "+ removeListener);
        print("====================");
      }
    }
    class Dumper implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        String name = query.getText();
        Class<?> c = null;
        try {
          c = Class.forName(name);
        } catch(ClassNotFoundException ex) {
          results.setText("Nie można znaleźć " + name);
          return;
        }
        dump(c);
      }
    }
    public BeanDumper() {
      JPanel p = new JPanel();
      p.setLayout(new FlowLayout());
      p.add(new JLabel("Kwalifikowana nazwa komponentu:"));
      p.add(query);
      add(BorderLayout.NORTH, p);
      add(new JScrollPane(results));
      Dumper dmpr = new Dumper();
      query.addActionListener(dmpr);
      query.setText("frogbean.Frog");
      // Wymuszenie operacji
      dmpr.actionPerformed(new ActionEvent(dmpr, 0, ""));
    }
    public static void main(String[] args) {
      run(new BeanDumper(), 600, 500);
    }
  } ///:~
```

Cała praca wykonywana jest przez metodę `BeanDumper.dump()`. Najpierw próbuje ona stworzyć obiekt `BeanInfo` i — jeśli się to powiedzie — wywołuje jego metody zwracające informacje o właściwości, metodach i zdarzeniach. Metoda `Introspector.getBeanInfo()` ma drugi argument. Mówi on obiektowi klasy `Introspector`, jak głęboko ma wejść w hierarchię dziedziczenia. Tutaj zatrzymuje się przed przetworzeniem metod klasy `Object`, ponieważ one nas nie interesują.

Dla właściwości metoda `getPropertyDescriptors()` zwraca tablicę obiektów `PropertyDescriptor`. Dla każdego z tych obiektów można wywołać metodę `getPropertyType()`, aby dowiedzieć się, jakiego typu obiekty są przyjmowane przez metody dostępu tej właściwości. Następnie dla każdej właściwości można przez `getName()` uzyskać jej nazwę (wydobyty z nazw metod), przez `getReadMethod()` metodę do odczytywania i przez `getWriteMethod()` metodę do zapisu. Te dwie ostatnie metody zwracają obiekt typu `Method`, którego można użyć do wywołania odpowiadającej mu metody wobec obiektu (jest to część mechanizmu refleksji).

Dla metod publicznych (wliczając metody dostępu do właściwości) `getMethodDescriptors()` zwraca tablicę obiektów `MethodDescriptor`. Dla każdego z nich można uzyskać związany z nim obiekt `Method` i wypisać jego nazwę.

Dla zdarzeń metoda `getEventSetDescriptors()` zwraca tablicę (a cóżby innego?) obiektów `EventSetDescriptor`. Każdy z nich może zostać zapytany o klasę odbiornika, metody dla odbiornika tej klasy oraz metody dodawania i usuwania odbiornika. Program `BeanDumper` wypisuje wszystkie te informacje.

Przy uruchomieniu program wymusza przepytanie `frogbean.Frog`. W wyniku, po usunięciu niepotrzebnych detali, otrzymujemy:

```
Typ właściwości:
  Color
Nazwa właściwości:
  color
Metoda odczytująca:
  public Color getColor()
Metoda ustawiająca:
  public void setColor(Color)
==================
Typ właściwości:
  boolean
Nazwa właściwości:
  jumper
Metoda odczytująca:
  public boolean isJumper()
Metoda ustawiająca:
  public void setJumper(boolean)
==================
Typ właściwości:
  int
Nazwa właściwości:
  jumps
Metoda odczytująca:
  public int getJumps()
Metoda ustawiająca:
  public void setJumps(int)
```

```
----------==========
Typ właściwości:
  frogbean.Spots
Nazwa właściwości:
  spots
Metoda odczytująca:
  public frogbean.Spots getSpots()
Metoda ustawiająca:
  public void setSpots(frogbean.Spots)
====================
Metody publiczne:
public void setSpots(frogbean.Spots)
public void setColor(Color)
public void setJumps(int)
public boolean isJumper()
public frogbean.Spots getSpots()
public void croak()
public void addActionListener(ActionListener)
public void addKeyListener(KeyListener)
public Color getColor()
public void setJumper(boolean)
public int getJumps()
public void removeActionListener(ActionListener)
public void removeKeyListener(KeyListener)
====================
Obsługa zdarzeń:
Typ odbiornika:
  KeyListener
Metoda odbiornika:
  keyPressed
Metoda odbiornika:
  keyReleased
Metoda odbiornika:
  keyTyped
Deskryptor metody:
  public abstract void keyPressed(KeyEvent)
Deskryptor metody:
  public abstract void keyReleased(KeyEvent)
Deskryptor metody:
  public abstract void keyTyped(KeyEvent)
Metoda dodająca odbiornik:
  public void addKeyListener(KeyListener)
Metoda usuwająca odbiornik:
  public void removeKeyListener(KeyListener)
--------------------
Typ odbiornika:
  ActionListener
Metoda odbiornika:
  actionPerformed
Deskryptor metody:
  public abstract void actionPerformed(ActionEvent)
Metoda dodająca odbiornik:
  public void addActionListener(ActionListener)
Metoda usuwająca odbiornik:
  public void removeActionListener(ActionListener)
====================
```

Widzimy tu większość informacji, jakie odnajduje Introspector, tworząc obiekt BeanInfo dla naszego komponentu JavaBean. Widać, że typ właściwości i jej nazwa są od siebie niezależne. Zwróć uwagę na zamianę pierwszej litery nazwy właściwości na małą (zamiana taka nie zachodzi tylko wtedy, gdy nazwa właściwości rozpoczyna się więcej niż jednym wielkim znakiem). Należy również pamiętać, że nazwy widocznych tu metod (jak metody odczytu i zapisu) są pobierane z obiektu typu Method, którego możemy użyć do wywołania powiązanej z nim metody obiektu komponentu.

Lista metod publicznych zawiera metody, które nie są powiązane z żadną właściwością czy zdarzeniem, jak np. croak(), ale również te, które są w taki sposób powiązane. Są to wszystkie metody, jakie można wywołać programowo wobec komponentu. Aby ułatwić pracę, narzędzia do budowy aplikacji mogą wyświetlać spis tych wszystkich metod, gdy dokonujemy wywołania metody z obiektu tego typu.

Na koniec widzimy, że wszystkie zdarzenia są sprowadzone do właściwego odbiornika, jego metod oraz metod dodawania i usuwania odbiorników. Kiedy już mamy obiekt BeanInfo, wiemy o kontrolce wszystko, co może być dla nas istotne. Możemy również wywoływać jej metody, nawet jeśli nie mamy żadnych innych informacji poza samym obiektem (przez mechanizm refleksji)[10].

Bardziej wyszukany komponent

Kolejny przykład jest odrobinę bardziej wyszukany, chociaż jednocześnie śmieszny. Jest to komponent typu JPanel, który rysuje małe kółko wokół wskaźnika myszy, gdziekolwiek zostanie on przesunięty. Kiedy zostanie naciśnięty przycisk myszy, na środku ekranu pojawi się słowo „Bum!" i nastąpi poinformowanie odbiornika zdarzenia.

Właściwości, które można zmieniać, to: rozmiar kółka, słowo, które ma być wyświetlane po naciśnięciu przycisku myszy, oraz jego kolor i rozmiar. Kontrolka BangBean posiada również własne metody addActionListener() i removeActionListener(), a więc można do niej podczepić jej odbiorniki, które zostaną poinformowane, kiedy użytkownik kliknie kontrolkę BangBean:

```
//: bangbean/BangBean.java
// Graficzny komponent JavaBean.
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

public class
BangBean extends JPanel implements Serializable {
  private int xm, ym;
  private int cSize = 20; // Rozmiar okręgu
  private String text = "Bum!";
```

[10] Mechanizm refleksji jest wykorzystywany tylko na etapie projektowania interfejsu z komponentów JavaBean w środowisku programistycznym. Kod odpowiedzialny za obsługę zdarzeń, wygenerowany przez środowisko, używa już statycznych wywołań metod — więc będzie wykonywał się szybko — dużo szybciej niż w przypadku refleksji, które wprowadzają dodatkowy narzut — *przyp. red.*

```
  private int fontSize = 48;
  private Color tColor = Color.RED;
  private ActionListener actionListener;
  public BangBean() {
    addMouseListener(new ML());
    addMouseMotionListener(new MML());
  }
  public int getCircleSize() { return cSize; }
  public void setCircleSize(int newSize) {
    cSize = newSize;
  }
  public String getBangText() { return text; }
  public void setBangText(String newText) {
    text = newText;
  }
  public int getFontSize() { return fontSize; }
  public void setFontSize(int newSize) {
    fontSize = newSize;
  }
  public Color getTextColor() { return tColor; }
  public void setTextColor(Color newColor) {
    tColor = newColor;
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.BLACK);
    g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
  }
  // To najprostszy do zarządzania odbiornik jednostkowy:
  public void addActionListener(ActionListener l)
  throws TooManyListenersException {
    if(actionListener != null)
      throw new TooManyListenersException();
    actionListener = l;
  }
  public void removeActionListener(ActionListener l) {
    actionListener = null;
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      Graphics g = getGraphics();
      g.setColor(tColor);
      g.setFont(
        new Font("TimesRoman", Font.BOLD, fontSize));
      int width = g.getFontMetrics().stringWidth(text);
      g.drawString(text, (getSize().width - width) /2,
        getSize().height/2);
      g.dispose();
      // Wywołanie metody odbiornika:
      if(actionListener != null)
        actionListener.actionPerformed(
          new ActionEvent(BangBean.this,
            ActionEvent.ACTION_PERFORMED, null));
    }
  }
  class MML extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
      xm = e.getX();
      ym = e.getY();
```

```
        repaint();
      }
    }
    public Dimension getPreferredSize() {
      return new Dimension(200, 200);
    }
  } ///:~
```

Pierwszą rzeczą, na którą należy zwrócić uwagę, jest to, że kontrolka BangBean implementuje interfejs Serializable. Oznacza to, że po tym, jak twórca programu ustawi wartości właściwości kontrolki, narzędzie do tworzenia aplikacji może zachować wszystkie te informacje, używając serializacji. Kiedy kontrolka jest później tworzona w działającej aplikacji, te zachowane właściwości są odtwarzane i odzyskujemy ustawione wcześniej wartości.

Patrząc na sygnaturę metody addActionListener(), możemy dostrzec, że może ona zgłosić wyjątek TooManyListenersException. Oznacza to, że jest to model *zdarzenia „jednostkowego"*, czyli o zajściu zdarzenia powiadomiony będzie tylko jeden odbiornik. Zazwyczaj używa się *rozgłaszania* zdarzenia tak, że o jego zajściu może zostać poinformowanych wiele odbiorników. Dochodzimy tu jednak do zagadnień związanych z wielowątkowością, którymi zajmiemy się dokładniej w dalszej części rozdziału, pt.: „Komponenty JavaBean i synchronizacja". Jak na razie rozwiązaniem problemu jest wykorzystanie modelu zdarzenia „jednostkowego".

Po naciśnięciu przycisku myszy na środku kontrolki BangBean pojawia się tekst i jeśli pole actionListener nie ma wartości null, tworzony będzie obiekt ActionEvent i wywoływana jego metoda actionPerformed(). Po każdym ruchu myszy przechwytywane są jej nowe współrzędne i odświeżana powierzchnia kontrolki (przez kasowanie tekstu, który był na niej).

Poniższa klasa BangBeanTest pozwala przetestować kontrolkę:

```
// : bangbean/BangBeanTest.java
// {Timeout: 5} Przerwanie po pięciu sekundach testu
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class BangBeanTest extends JFrame {
  private JTextField txt = new JTextField(20);
  // Zgłaszanie czynności w czasie testowania:
  class BBL implements ActionListener {
    private int count = 0;
    public void actionPerformed(ActionEvent e) {
      txt.setText("BangBean czynność "+ count++);
    }
  }
  public BangBeanTest() {
    BangBean bb = new BangBean();
    try {
      bb.addActionListener(new BBL());
    } catch(TooManyListenersException e) {
      txt.setText("Za dużo odbiorników");
```

```
    }
    add(bb);
    add(BorderLayout.SOUTH, txt);
  }
  public static void main(String[] args) {
    run(new BangBeanTest(), 400, 500);
  }
} ///:~
```

Powyższy program jest zbędny, gdy komponent jest używany w środowisku programistycznym. Mimo to jest przydatny jako sposób na szybkie przetestowanie tworzonych elementów. Klasa `BangBeanTest` umieszcza kontrolkę `BangBean` na formatce i dołącza do niej prosty odbiornik `ActionListener`, który wyświetla liczbę wystąpień zdarzenia. Oczywiście przeważnie to narzędzie do tworzenia aplikacji generuje większość kodu wykorzystującego komponent JavaBean.

Po przepuszczeniu kontrolki `BangBean` przez wcześniej przedstawiony program `BeanDumper` lub po umieszczeniu jej w środowisku programistycznym obsługującym komponenty przekonamy się, że posiada ona o wiele więcej atrybutów i zdarzeń, niż wynikałoby to z powyższego kodu. Jest tak, ponieważ `BangBean` dziedziczy po klasie `JPanel`, która także jest komponentem JavaBean, toteż widzimy również jej właściwości i zdarzenia.

Ćwiczenie 35. Odszukaj i ściągnij z Internetu chociaż jedno z darmowych środowisk programistycznych lub skorzystaj z posiadanego produktu komercyjnego. Sprawdź, co należy zrobić, by móc w nim używać komponentu `BangBean` i zrób to (6).

Komponenty JavaBean i synchronizacja

Tworząc komponenty JavaBean, należy założyć, że będą one używane w środowiskach wielowątkowych. Oznacza to że:

1. Jeśli to tylko możliwe, wszystkie publiczne metody komponentu powinny być synchronizowane. Oczywiście wiąże się to z narzutami czasowymi związanymi z synchronizacją (które na szczęście zostały znacząco zredukowane w najnowszych wersjach JDK). Jeśli synchronizacja jest problemem, to metod, które nie będą przysparzać problemów w krytycznej sekcji kodu, można nie synchronizować, należy jednak pamiętać, że rozwiązanie to nie zawsze jest oczywiste. Metody, które można pozostawić bez synchronizacji, są zazwyczaj małe (takie jak `getCircleSize()`) bądź „atomowe" — co oznacza, że są na tyle krótkie, iż podczas ich wykonywania obiekt nie może ulec modyfikacji (zajrzyj jednak do rozdziału „Współbieżność" — przypomnisz sobie, że nie zawsze to, co wygląda na atomowe, jest faktycznie takie). Tyle że rezygnacja z synchronizowania takich metod nie będzie mieć większego wpływu na efektywność działania programu. Lepiej byłoby więc z założenia synchronizować wszystkie publiczne metody komponentu i usuwać słowo kluczowe `synchronized` wyłącznie w przypadkach, gdy jesteśmy pewni, że nic jest ono konieczne, a jego usunięcie ma istotne znaczenie dla wydajności.
2. Rozgłaszając zdarzenia do wielu odbiorników, koniecznie należy założyć, że w tym czasie mogą zostać dodane nowe odbiorniki, a te wcześniej zarejestrowane — mogą zostać usunięte.

Z zagadnieniami opisanymi w pierwszym z powyższych punktów można sobie poradzić w miarę łatwo, jednak drugi z nich wymaga dokładniejszego przemyślenia. W poprzedniej wersji programu *BangBean.java* problem współbieżności ominęliśmy poprzez zignorowanie słowa kluczowego synchronized i wykorzystanie zdarzeń jednostkowych. Oto nowa wersja programu, która może działać w środowiskach wielowątkowych i wykorzystuje rozgłaszanie zdarzeń:

```
// : gui/BangBean2.java
// Komponenty JavaBean należy pisać tak, aby były
// przygotowane do działania w środowisku wielowątkowym.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class BangBean2 extends JPanel
implements Serializable {
  private int xm, ym;
  private int cSize = 20; // Rozmiar okręgu
  private String text = "Bum!";
  private int fontSize = 48;
  private Color tColor = Color.RED;
  private ArrayList<ActionListener> actionListeners =
    new ArrayList<ActionListener>();
  public BangBean2() {
    addMouseListener(new ML());
    addMouseMotionListener(new MM());
  }
  public synchronized int getCircleSize() { return cSize; }
  public synchronized void setCircleSize(int newSize) {
    cSize = newSize;
  }
  public synchronized String getBangText() { return text; }
  public synchronized void setBangText(String newText) {
    text = newText;
  }
  public synchronized int getFontSize(){ return fontSize; }
  public synchronized void setFontSize(int newSize) {
    fontSize = newSize;
  }
  public synchronized Color getTextColor(){ return tColor;}
  public synchronized void setTextColor(Color newColor) {
    tColor = newColor;
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.BLACK);
    g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
  }
  // Odbiornik rozgłoszeniowy, bardziej typowy od odbiornika
  // jednostkowego zastosowanego w komponencie BangBean.java:
  public synchronized void
  addActionListener(ActionListener l) {
    actionListeners.add(l);
  }
```

```
  public synchronized void
  removeActionListener(ActionListener l) {
    actionListeners.remove(l);
  }
  // Uwaga na brak synchronizacji:
  public void notifyListeners() {
    ActionEvent a = new ActionEvent(BangBean2.this,
      ActionEvent.ACTION_PERFORMED, null);
    ArrayList<ActionListener> lv = null;
    // Utworzenie płytkiej kopii listy na wypadek,
    // gdyby ktoś dodał nowy odbiornik w trakcie
    // wywoływania odbiorników:
    synchronized(this) {
      lv = new ArrayList<ActionListener>(actionListeners);
    }
    // Wywołanie metod odbiorników:
    for(ActionListener al : lv)
      al.actionPerformed(a);
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      Graphics g = getGraphics();
      g.setColor(tColor);
      g.setFont(
        new Font("TimesRoman", Font.BOLD, fontSize));
      int width = g.getFontMetrics().stringWidth(text);
      g.drawString(text, (getSize().width - width) /2,
        getSize().height/2);
      g.dispose();
      notifyListeners();
    }
  }
  class MM extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
      xm = e.getX();
      ym = e.getY();
      repaint();
    }
  }
  public static void main(String[] args) {
    BangBean2 bb2 = new BangBean2();
    bb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("ActionEvent" + e);
      }
    });
    bb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("BangBean2 akcja");
      }
    });
    bb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("Kolejna akcja");
      }
    });
    JFrame frame = new JFrame();
```

```
    frame.add(bb2);
    run(frame, 300, 300);
  }
} ///:~
```

Dodanie słowa kluczowego synchronized do metody jest łatwą modyfikacją. Należy jednak zwrócić uwagę na metody addActionListener() oraz removeActionListener(), w których odbiorniki ActionListener są teraz dodawane i usuwane z listy ArrayList, a zatem może ich być dowolnie dużo.

Jak widać, metoda notifyListeners() *nie* jest synchronizowana. W dowolnej chwili można ją zatem wywołać z dowolnej liczby wątków. Istnieje także możliwość wywołania metod addActionListener() oraz removeActionListener() podczas wykonywania metody notifyListeners(), co jest pewnym problemem, gdyż metoda ta operuje na liście actionListeners. W celu rozwiązania tego problemu lista ArrayList jest „klonowana" wewnątrz klauzuli synchronized, a metoda operuje nie na liście oryginalnej, lecz na jej kopii (szczegółowe informacje na temat klonowania obiektów można znaleźć w suplementach towarzyszących książce). Dzięki temu można modyfikować oryginalną listę ArrayList bez żadnych negatywnych skutków dla działania metody notifyListeners().

Także metoda paintComponent() nie jest synchronizowana. Podjęcie decyzji, czy należy synchronizować przesłaniane metody czy nie, nie jest równie oczywiste jak w przypadku samodzielnie definiowanych metod. W powyższym przykładzie okazuje się, że metoda paintComponent() działa dobrze niezależnie do tego, czy jest synchronizowana czy nie. Jednak podejmując tę decyzję, należy wziąć pod uwagę następujące czynniki:

1. Czy metoda zmienia stan „krytycznych" zmiennych obiektu? Aby określić, czy zmienne są „krytyczne", należy sprawdzić, czy są one odczytywane bądź ustawiane przez inne wątki programu. (W takich przypadkach operacje odczytu i zapisu są niemal zawsze realizowane przez metody synchronizowane, a zatem wystarczy sprawdzić, czy są takie metody.) W metodzie paintComponent() nie są wykonywane żadne modyfikacje.
2. Czy działanie metody zależy od stanu zmiennych „krytycznych"? Jeśli synchronizowana metoda zmienia stan zmiennej używanej przez inną metodę, to w wielu przypadkach należy synchronizować także tę drugą metodę. Bazując na tej zasadzie, można zauważyć, że zmienna cSize jest modyfikowana przez metody synchronizowane, a zatem także metoda paintComponent() powinna być synchronizowana. W naszym przypadku można sobie jednak zadać pytanie: „Jaka jest najgorsza rzecz, która może się zdarzyć, gdy wartość cSize zostanie zmieniona w czasie realizacji metody paintComponent()?". Kiedy okaże się, że nic strasznego nie może się stać, można nie synchronizować metody paintComponent() i nie narażać się tym samym na niepotrzebne narzuty czasowe.
3. Kolejną podpowiedzią może być fakt, czy metoda paintComponent() w klasie bazowej jest synchronizowana czy nie. Okazuje się, że nie jest. Nie jest to jednak żaden ważki argument, a jedynie podpowiedź. W naszym przypadku pole *modyfikowane* przez synchronizowane metody (czyli cSize) jest używane w wyrażeniu wyliczanym w metodzie paintComponent(), co może zmienić sytuację. Warto jednak zwrócić uwagę, że fakt synchronizowania metody nie jest dziedziczony; oznacza to, że jeśli metoda jest synchronizowana w klasie bazowej, to *nie* będzie automatycznie synchronizowana w klasie potomnej.

4. Metody `paint()` oraz `paintComponent()` muszą działać możliwie jak najszybciej. Wszelkie sposoby przyspieszenia ich działania są jak najbardziej pożądane, a zatem konieczność ich synchronizacji może oznaczać błędny projekt programu.

W porównaniu z programem `BangBeanTest` testowy kod umieszczony w metodzie `main()` został zmodyfikowany, aby zademonstrować możliwości rozgłaszania zdarzeń do wielu odbiorników, jakimi dysponuje klasa `BangBean2`.

Pakowanie komponentu JavaBean

Zanim będziemy mogli umieścić nasz komponent w przygotowanym na technologię JavaBeans środowisku programistycznym, musi on zostać zapakowany w standardowy pojemnik dla komponentów. Jest nim plik JAR (Java ARchive), zawierający wszystkie klasy kontrolki oraz plik „manifestu" mówiący, że jest to komponent JavaBean. Plik manifestu to po prostu plik tekstowy szczególnej postaci. Dla naszej kontrolki `BangBean` plik taki wygląda następująco:

```
Manifest-Version: 1.0

Name: bangbean/BangBean.class
Java-Bean: True
```

Pierwszy wiersz oznacza wersję tego schematu manifestu, który pozostanie w wersji 1.0, dopóki nie doczeka się większej uwagi ze strony firmy Sun. Drugi wiersz (puste są ignorowane) określa plik *BangBean.class*, a trzeci mówi: „To jest komponent JavaBean". Bez tego trzeciego wiersza narzędzia programistycznie nie rozpoznałyby klasy jako kontrolki JavaBean.

Jedyną „podstępną" częścią jest umieszczenie w polu „Name:" właściwej ścieżki do pliku klasy. Jeśli przyjrzymy się ponownie klasie `BangBean`, zauważymy, że jest ona umieszczona w pakiecie `bangbean` (a zatem w podkatalogu o nazwie *bangbean*, który nie znajduje się na ścieżkach ustalonych przez zmienną `CLASSPATH`), toteż nazwa w manifeście musi zawierać informacje o tym pakiecie. Ponadto plik manifestu musi się znajdować w katalogu *nadrzędnym* dla ścieżki pakietu, co w naszym przypadku oznacza umieszczenie tego pliku w katalogu nadrzędnym wobec podkatalogu *bangbean*. Następnie należy wykonać poniższe polecenie w tym samym katalogu, co plik manifestu:

```
jar cfm BangBean.jar BangBean.mf bangbean
```

Zakładamy, że plikiem wynikowym ma być *BangBean.jar*, a manifest znajduje się w pliku o nazwie *BangBean.mf*.

Można się jeszcze zastanawiać: „A co z klasami, które były wygenerowane podczas kompilacji *BangBean.java*?". Znajdują się one wszystkie w podkatalogu, a — jak widać wcześniej — ostatnim parametrem wywołania *jar* jest podkatalog *bangbean*. Jeśli poda się programowi *jar* nazwę podkatalogu, to pakuje on do pliku JAR cały podkatalog (w naszym przypadku łącznie z oryginalnym plikiem źródłowym *BangBean.java* — przy własnych komponentach możesz zadecydować, by nie był pakowany kod źródłowy). Jeśli teraz rozpakowalibyśmy stworzony właśnie plik JAR, okazałoby się, że znajdujący się tam plik manifestu nie jest już tym samym plikiem, który stworzyliśmy. Program *jar* stworzył swoją własną wersję pliku manifestu (na podstawie naszego) o nazwie *MANIFEST.MF*

i umieścił go w podkatalogu *META-INF* (od metainformacja). Oglądając ten plik, można zobaczyć, że dla każdego pliku dodanego przez *jar* został tam umieszczony cyfrowy podpis postaci:

```
Digest-Algorithms: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/OO=
MD5-Digest: O4NcSlhE3Smnzlp2hj6qeg==
```

Generalnie nie ma potrzeby, żeby się tym przejmować i jeśli wprowadzimy jakiekolwiek zmiany, to w celu stworzenia nowego pliku JAR wystarczy tylko zmodyfikować oryginalny plik manifestu i ponownie wywołać program *jar*. W tym samym pliku JAR można umieścić kolejne komponenty, po prostu dodając ich informacje do manifestu.

Na jedną rzecz należy zwrócić uwagę. Prawdopodobnie będziemy chcieli umieścić każdą kontrolkę we własnym podkatalogu. Kiedy tworząc plik JAR, przekażemy narzędziu *jar* nazwę podkatalogu, to do archiwum JAR zostanie włączona cała zawartość tego katalogu. Można się przekonać, że klasy `Frog` i `BangBean` są w swoich własnych podkatalogach.

Kiedy już mamy kontrolkę JavaBean prawidłowo spakowaną w pliku JAR, można jej użyć w środowisku programistycznym. Sposób, w jaki się to robi, jest zależny od konkretnego narzędzia. Jednak firma Sun dostarcza bezpłatne narzędzie do testowania kontrolek JavaBean o nazwie *Bean Builder* (można je pobrać spod adresu *http://java.sun.com/beans*). Aby umieścić komponent w środowisku Bean Builder, wystarczy przed uruchomieniem tego programu skopiować plik JAR do odpowiedniego podkatalogu.

Ćwiczenie 36. Dodaj plik *Frog.class* do pliku manifestu i uruchom program *jar*, by utworzyć plik archiwum zawierający klasy `Frog` i `BangBean`. Teraz ściągnij z internetu i zainstaluj program *Bean Builder* firmy Sun albo użyj własnego narzędzia obsługującego komponenty JavaBean i dodaj do niego plik JAR, by móc przetestować obydwa komponenty (4).

Ćwiczenie 37. Utwórz własną kontrolkę JavaBean o nazwie `Valve` (zawór) posiadającą dwa atrybuty: typu `boolean` o nazwie „włączony" oraz liczbową, typu `int` o nazwie „poziom". Utwórz plik manifestu i użyj narzędzia *jar* do spakowania kontrolki, a następnie załaduj ją do programu *Bean Builder* lub środowiska programistycznego obsługującego kontrolki JavaBean i przetestuj ją (5).

Bardziej złożona obsługa komponentów JavaBean

Widać już, jak niesamowicie proste jest tworzenie komponentów JavaBean. Jednak nie ogranicza się to do tego, co tu pokazałem. Architektura JavaBeans daje dobry punkt zaczepienia, ale można ją też stosować w bardziej złożonych sytuacjach. Sytuacje te wykraczają poza zakres tej książki, ale zostaną tu pokrótce omówione. Więcej informacji można znaleźć pod adresem *http://java.sun.com/beans*.

Jednym z miejsc, w których można dodać bardziej wyszukane mechanizmy, są właściwości. Prezentowane wcześniej przykłady zawierały jedynie pojedyncze właściwości, ale można również reprezentować wielokrotne właściwości w tablicy. Nazywa się je wtedy *właściwościami indeksowanymi*. Wystarczy jedynie dodać odpowiednie metody (zgodnie z przyjętą konwencją nazewniczą), aby `Introspector` rozpoznał właściwość indeksowaną i narzędzie do tworzenia aplikacji powinno już odpowiednio się zachować.

Właściwości mogą być *związane* (ang. *bound*), co oznacza, że inne obiekty będą informowane o ich zmianie przez zdarzenie `PropertyChangeEvent`. Powiadomione obiekty mogą wtedy zmienić swój stan, wykorzystując modyfikację komponentu JavaBean.

Właściwości mogą być *ograniczone* (ang. *constrained*), co oznacza, że inne obiekty mogą zawetować zmianę takiej właściwości, jeśli jest ona nie do zaakceptowania. Obiekty są powiadamiane poprzez zdarzenie `PropertyChangeEvent`. Aby powstrzymać zmianę i wrócić do poprzedniej wartości, mogą wtedy same zgłosić wyjątek `PropertyVetoException`.

Można również zmienić sposób reprezentacji komponentu w czasie projektowania:

1. Można stworzyć własny arkusz właściwości dla konkretnej kontrolki. Zwyczajny arkusz właściwości będzie używany dla wszystkich innych kontrolek, a nasz wywoływany automatycznie, jeśli zostanie wybrany nasz komponent.
2. Można stworzyć własny edytor dla konkretnej właściwości. Używany jest nadal zwyczajny arkusz właściwości, ale kiedy edytowana jest taka specjalna właściwość, to automatycznie wywoływany będzie nasz edytor.
3. Można stworzyć własną klasę `BeanInfo` dla komponentu, zwracającą inne informacje niż uzyskiwane domyślnie za pomocą obiektu klasy `Introspector`.
4. Jest również możliwe włączenie lub wyłączenie w `FeatureDescriptor` trybu „eksperckiego", aby odróżnić cechy podstawowe od bardziej skomplikowanych.

Więcej o komponentach JavaBean

Dostępnych jest też wiele książek na temat JavaBeans, na przykład *JavaBeans* autorstwa Elliotte'a Rusty Harolda (IDG, 1998).

Alternatywy wobec biblioteki Swing

Choć Swing jest biblioteką interfejsu graficznego uświęconą przez firmę Sun, nie jest bynajmniej jedyną taką biblioteką. Wśród alternatyw są przynajmniej dwie poważne: Macromedia Flash, z systemem programistycznym Macromedia Flex służącym do tworzenia interfejsów strony klienckiej aplikacji WWW, oraz otwarta biblioteka elementów interfejsu graficznego środowiska Eclipse, znana pod nazwą SWT (*Standard Widget Toolkit*), a przeznaczona dla aplikacji stacjonarnych.

Po cóż nam jakiekolwiek alternatywy? Otóż nie sposób nie zauważyć porażki apletów na polu klienckich aplikacji WWW. Jeśli wziąć pod uwagę to, jak długo znajdują się w użyciu (od początku istnienia języka Java) oraz to, ile było wokół nich szumu i ile z nimi wiązano nadziei, liczba aplikacji WWW faktycznie korzystających z apletów po stronie klienta jest doprawdy mizerna. Nawet Sun nie wykorzystuje apletów wszędzie tam, gdzie to możliwe. Oto dowód:

http://java.sun.com/developer/onlineTraining/new2java/javamap/intro.html

Publikowana tam interaktywna mapa funkcji Javy wydaje się świetnym kandydatem na aplet, mimo to jest to prezentacja Flash. To chyba ostateczne przyznanie się do porażki koncepcji apletów. Co więcej, odtwarzacz prezentacji Flash jest instalowany na 98 procentach wszystkich platform komputerowych, przez co stanowi de facto standard. Przekonasz się też, że system Flex to bardzo efektywny system programistyczny strony klienckiej aplikacji WWW, z pewnością efektywniejszy od JavaScriptu; do tego wygląd interfejsów Flash dalece przewyższa możliwości apletów. Tam, gdzie trzeba używać tych ostatnich, należy zadbać o pobranie i zainstalowanie dość obszernych bibliotek JRE; dla porównania, odtwarzacze prezentacji Flash są zazwyczaj zwarte, przez co ich ewentualne pobieranie jest znacznie mniej kłopotliwe.

Co do aplikacji stacjonarnych, to Swing ma tę wadę, że użytkownicy łatwo *zauważają* fakt korzystania z aplikacji odmiennego rodzaju — wszystko z powodu stylu wyglądu elementów interfejsu użytkownika, zdecydowanie odmiennego od stylu wyglądu tradycyjnych aplikacji. Odmiana w zakresie stylu interfejsu nie jest wcale mile widziana przez użytkowników — ci zazwyczaj skupiają się na swoich zadaniach i wolą takie aplikacje, których wygląd jest dla nich możliwie swojski. Biblioteka SWT pozwala na tworzenie aplikacji z interfejsem maksymalnie upodobnionym do macierzystych aplikacji dla danego systemu, a do tego działających szybciej niż ich odpowiedniki korzystające z biblioteki Swing.

Flex — aplikacje klienckie w formacie Flash

Z racji powszechności wirtualnej maszyny Macromedia Flash aplikacje w formacie Flash są dostępne dla szerokiej rzeszy użytkowników, bez konieczności instalowania dodatkowego oprogramowania; do tego aplikacje Flash wyglądają tak samo niezależnie od systemu, w którym zostaną uruchomione[11].

Dzięki systemowi *Macromedia Flex* można tworzyć flashowe interfejsy użytkownika dla aplikacji języka Java. Flex składa się z modelu programistycznego opartego na XML-u i skryptach, podobnego do modelu programistycznych takich języków jak HTML i JavaScript, oraz solidnego zestawu komponentów. Do deklarowania układu interfejsu i wyglądu jego elementów służy składnia MXML, a za pomocą dynamicznych skryptów oprogramowuje się obsługę zdarzeń i kodu wywołującego usługi łączące użytkownika z klasami Javy, modelami danych, usługami WWW i tym podobnymi. Kompilator Flex operuje na opisie MXML i plikach skryptów i na ich podstawie tworzy kod bajtowy, interpretowany potem w maszynie wirtualnej Flasha działającej w systemie klienta. Format bajtowy Flash nosi nazwę SWF — kompilator Flex tworzy właśnie pliki SWF.

Flex doczekał się swojego wolnego (w znaczeniu *open-source*) odpowiednika o podobnej strukturze — zobacz *http://openlaszlo.org*; dla niektórych może to być całkiem pociągająca alternatywa. Istnieją też różne inne narzędzia do tworzenia aplikacji Flash.

[11] Materiał do tego podrozdziału przygotował Sean Neville.

Ahoj, Flex

Spójrz na poniższy kod MXML, definiujący interfejs użytkownika (pierwszy i ostatni wiersz listingu nie stanowią właściwego kodu i nie występują w plikach źródłowych rozprowadzanych wraz z książką):

```
//:! gui/flex/helloflex1.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.macromedia.com/2003/mxml"
  backgroundColor="#ffffff">
  <mx:Label id="output" text="Ahoj, Flex!" />
</mx:Application>
//:~
```

Pliki MXML są dokumentami języka XML, więc zaczynają się od deklaracji XML wersji i kodowania. Skrajnie zewnętrzny element MXML to `Application`, reprezentujący główny wizualny i logiczny kontener interfejsu użytkownika Flex. W jego obrębie deklaruje się znaczniki reprezentujące kontrolki wizualne, jak etykieta `Label` w powyższym przykładzie. Kontrolki są zawsze rozmieszczane w obrębie kontenera, a kontenery do zarządzania układem kontrolek wykorzystują między innymi menedżery układu. W najprostszym przypadku (takim jak powyższy) rolę kontenera pełni `Application`. Domyślny menedżer układu kontenera `Application` rozmieszcza kontrolki w pionie, w kolejności zgodnej z kolejnością deklaracji.

ActionScript to odmiana ECMAScriptu, albo JavaScriptu, dość upodobniona do Javy i obsługująca klasy i ścisłą kontrolę typów. Uzupełniając przykład skryptem, ożywiamy aplikację. Do osadzenia skryptu w pliki MXML można wykorzystać element `Script`:

```
//:! gui/flex/helloflex2.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.macromedia.com/2003/mxml"
  backgroundColor="#ffffff">
  <mx:Script>
    <![CDATA[
    function updateOutput() {
      output.text = "Ahoj! " + input.text;
    }
    ]]>
  </mx:Script>
  <mx:TextInput id="input" width="200"
    change="updateOutput()" />
  <mx:Label id="output" text="Ahoj!" />
</mx:Application>
//:~
```

Kontrolka `TextInput` służy do wprowadzania danych przez użytkownika, a etykieta (`Label`) na bieżąco wypisuje tekst wprowadzany do kontrolki. Zauważ, że w obrębie skryptu atrybut `id` (identyfikator) każdej kontrolki staje się dostępny jako najzwyklejsza zmienna, przez co w skrypcie można odwoływać się do poszczególnych egzemplarzy znaczników. Atrybut `change` pola `TextInput` został podłączony do funkcji `updateOutput()`, co daje efekt wywołania tejże funkcji przy każdej zmianie zawartości pola tekstowego.

Kompilowanie MXML

Aby zacząć wykorzystywanie systemu Flex, najlepiej skorzystać z darmowej próbki publikowanej pod adresem *www.macromedia.com/software/flex/trial*[12]. Produkt jest dostępny w rozmaitych edycjach, od wersji próbnych po serwerowe wersje korporacyjne, a sama firma Macromedia oferuje dodatkowe narzędzia do opracowywania aplikacji Flex. Sposób pakowania może się zmieniać, więc w poszukiwaniu szczegółów będziesz musiał zapoznać się z treścią podanej strony. Pamiętaj też o potencjalnej konieczności zmiany pliku *jvm.config* umieszczonego w podkatalogu *bin* instalacji Flex.

Kompilacja kodu MXML do postaci kodu bajtowego Flash może się odbyć dwojako:

1. Poprzez umieszczenie pliku MXML w aplikacji WWW Javy, w pliku WAR wraz ze stronami JSP oraz HTML i kompilację odwołań do *.mxml* w czasie wykonania, w reakcji na żądanie przeglądarki kierowane do pliku MXML.
2. Poprzez ręczne wywołanie kompilatora Flex (`mxmlc`) z poziomu wiersza poleceń.

Opcja pierwsza, czyli kompilacja dynamiczna za pośrednictwem serwera WWW, wymaga uzupełnienia systemu Flex o kontener serwletów (taki jak Apache Tomcat). Plik (pliki) WAR kontenera serwletów musi być uzupełniony informacjami konfiguracyjnymi systemu Flex, a więc odwzorowaniami dodawanymi do deskryptora *web.xml*, i musi obejmować pliki JAR systemu Flex — tym zajmuje się sam system automatycznie, przy instalacji. Po skonfigurowaniu pliku WAR można umieszczać pliki MXML w aplikacjach WWW i odwoływać się do dokumentów aplikacji Flex za pośrednictwem przeglądarki. Przy pierwszym odwołaniu Flex skompiluje aplikację (podobnie, jak w modelu JSP) i dostarczy do przeglądarki skompilowany i zbuforowany plik SWF w otoczce dokumentu HTML.

Druga opcja nie wymaga udziału serwera. Wywołanie kompilatora Flex działającego w wierszu poleceń tworzy potrzebne pliki SWF. Pliki te można potem instalować wedle uznania. Program kompilatora `mxmlc` znajduje się w podkatalogu *bin* katalogu instalacyjnego systemu Flex; wywołany bez argumentów wyświetli listę rozpoznawanych opcji. Zazwyczaj w wywołaniu podaje się położenie biblioteki komponentów Flex (służy do tego opcja `-flexlib`), ale w najprostszych przykładach, jak dwa poprzednie, można polegać na domyślnym położeniu biblioteki komponentów. Oba poprzednie przykłady można skompilować tak:

```
mxmlc.exe helloflex1.mxml
mxmlc.exe helloflex2.mxml
```

To drugie polecenie utworzy plik *helloflex2.swf*, nadający się do uruchomienia w odtwarzaczu formatu Flash albo do umieszczenia w dokumencie HTML na dowolnym serwerze protokołu HTTP (po załadowaniu odtwarzacza Flash do przeglądarki wystarczy dwukrotnie kliknąć odnośnik SWF w celu uruchomienia aplikacji w przeglądarce).

[12] Trzeba stamtąd pobrać system Flex, a nie FlexBuilder. Ten ostatni to kompletne zintegrowane środowisko programistyczne.

W przypadku pliku *helloflex2.swf* w odtwarzaczu Flash pojawi się interfejs podobny do poniższego:

Ahoj! To nie było takie trudne...

W bardziej rozbudowanych aplikacjach plik MXML można oddzielić od skryptu ActionScript, odwołując się do funkcji zapisanych w zewnętrznych plikach ActionScript. W plikach MXML używa się wtedy następującej składni elementów Script:

```
<mx:Script source="ZewnetrznySkrypt.as" />
```

Taki kod pozwala kontrolce MXML odwoływać się do funkcji umieszczonych w pliku o nazwie *ZewnetrznySkrypt.as*, tak jakby te funkcje były umieszczone wprost w pliku MXML.

MXML i skrypty ActionScript

MXML to deklaratywny skrót dla klas skryptów ActionScript. Każdemu znacznikowi MXML odpowiada klasa ActionScript o identycznej nazwie. Kiedy kompilator Flex przetwarza plik MXML, najpierw tłumaczy dokument XML na kod ActionScript, wczytuje wymieniane klasy ActionScript, po czym kompiluje kod do postaci pliku SWF.

Na upartego można by całą aplikację Flex napisać w czystym kodzie ActionScript, bez uciekania się do dokumentów MXML. MXML służy jedynie wygodzie programisty. Elementy interfejsu użytkownika, takie jak kontenery i kontrolki, są zazwyczaj deklarowane za pomocą MXML, podczas gdy logika aplikacji, taka jak kod obsługi zdarzeń i tym podobne są oprogramowywane w Javie i skryptach ActionScript.

Programista może też tworzyć własne kontrolki MXML i odwoływać się do własnych klas ActionScript. Można też zebrać istniejące kontenery i kontrolki MXML w nowym dokumencie MXML, który potem może występować jako znacznik w innym dokumencie MXML. Więcej informacji na ten temat zawiera strona WWW firmy Macromedia.

Kontenery i kontrolki

Rdzeniem wizualnym biblioteki komponentów Flex jest zestaw kontenerów zarządzających układem zawartości, w których przechowywane są tablice kontrolek. Do kontenerów zaliczamy panele, obszary prostokątne pionowe i poziomie, klocki, harmonijki (ang. *accordion*), pudełka dzielone, siatki i inne. Do kontrolek należą zaś właściwe elementy interfejsu użytkownika, jak przyciski, pola tekstowe, suwaki, kalendarze, siatki danych i tak dalej.

W pozostałej części niniejszego rozdziału zajmiemy się aplikacją Flex wyświetlającą i porządkującą listę plików dźwiękowych. Aplikacja ta zademonstruje stosowanie kontenerów i kontrolek oraz sposoby łączenia Flasha z aplikacjami języka Java.

Zaczniemy od pliku MXML z kontrolką DataGrid (jedną z bardziej skomplikowanych kontrolek systemu Flex) w kontenerze Panel:

```
//:! gui/flex/songs.mxml
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.macromedia.com/2003/mxml"
  backgroundColor="#B9CAD2" pageTitle="Kolekcjoner utworów Flex"
  initialize="getSongs()">
  <mx:Script source="songScript.as" />
  <mx:Style source="songStyles.css"/>
  <mx:Panel id="songListPanel"
    titleStyleDeclaration="headerText"
    title="Biblioteka MP3 (Flex)">
    <mx:HBox verticalAlign="bottom">
      <mx:DataGrid id="songGrid"
        cellPress="selectSong(event)" rowCount="8">
        <mx:columns>
          <mx:Array>
            <mx:DataGridColumn columnName="name"
              headerText="Nazwa utworu" width="120" />
            <mx:DataGridColumn columnName="artist"
              headerText="Wykonawca" width="180" />
            <mx:DataGridColumn columnName="album"
              headerText="Album" width="160" />
          </mx:Array>
        </mx:columns>
      </mx:DataGrid>
      <mx:VBox>
        <mx:HBox height="100" >
          <mx:Image id="albumImage" source=""
            height="80" width="100"
            mouseOverEffect="resizeBig"
            mouseOutEffect="resizeSmall" />
          <mx:TextArea id="songInfo"
            styleName="boldText" height="100%" width="120"
            vScrollPolicy="off" borderStyle="none" />
        </mx:HBox>
        <mx:MediaPlayback id="songPlayer"
          contentPath=""
          mediaType="MP3"
          height="70"
          width="230"
          controllerPolicy="on"
          autoPlay="false"
          visible="false" />
      </mx:VBox>
    </mx:HBox>
    <mx:ControlBar horizontalAlign="right">
      <mx:Button id="refreshSongsButton"
        label="Odśwież" width="100"
        toolTip="Odświeża listę utworów"
        click="songService.getSongs()" />
    </mx:ControlBar>
  </mx:Panel>
  <mx:Effect>
    <mx:Resize name="resizeBig" heightTo="100"
      duration="500"/>
```

```
    <mx:Resize name="resizeSmall" heightTo="80"
      duration="500"/>
  </mx:Effect>
  <mx:RemoteObject id="songService"
    source="gui.flex.SongService"
    result="onSongs(event.result)"
    fault="alert(event.fault.faultstring, 'Błąd')">
    <mx:method name="getSongs"/>
  </mx:RemoteObject>
</mx:Application>
//:~
```

`DataGrid` zawiera zagnieżdżone znaczniki dla tablicy kolumn. Atrybut albo zagnieżdżony element kontrolki oznacza jakąś jej właściwość, zdarzenie bądź osadzony obiekt klasy ActionScript. Kontrolka `DataGrid` posiada atrybut `id` z wartością `songGrid`, więc kod ActionScript i znaczniki MXML mogą się do niej odwoływać programowo za pośrednictwem zmiennej `songGrid`. Sama kontrolka `DataGrid` eksponuje znacznie więcej właściwości, niż zostało użytych w przykładzie; kompletny interfejs programistyczny kontrolek i kontenerów MXML został udokumentowany w publikacji spod adresu *http://livedocs.macromedia.com/flex/15/asdocs_en/index.html*.

Kontrolka `DataGrid` jest uzupełniana kontrolką `VBox` zawierającą obrazek (`Image`) prezentujący okładkę płyty i informacje o utworze, oraz kontrolkę `MediaPlayback` służącą do inicjowania odsłuchu utworów MP3. W tym przykładzie odsłuchiwane utwory są odtwarzane strumieniowo, co redukuje rozmiar skompilowanego pliku SWF. Jeśli w aplikacji SWF osadzamy wprost obrazki oraz pliki dźwiękowe i wideo, zamiast je strumieniować, pliki te stają się integralną częścią aplikacji i są rozprowadzane wraz z interfejsem użytkownika, zwiększając znacząco jego rozmiar (w porównaniu do modelu korzystającego z odtwarzania strumieniowego).

Odtwarzacz Flash Player zawiera własne kodeki do odtwarzania strumieni audio i wideo rozmaitych formatów. Flash i Flex obsługują też najpopularniejsze formaty graficzne wykorzystywane na stronach WWW, ponadto Flex ma możliwość tłumaczenia grafiki wektorowej SVG na zasoby SWF nadające się do osadzania w aplikacjach klienckich Flex.

Efekty i style

Odtwarzacz Flash Player wyświetla grafikę wektorowo, może więc wydajnie wykonywać przekształcenia w czasie wykonania aplikacji. Przedsmak możliwych do osiągnięcia animacji dają *efekty* systemu Flex. Efekty są transformacjami stosowanymi wobec kontrolek i kontenerów z poziomu składni MXML.

Znacznik `Effect` widniejący w dokumencie MXML daje dwojaki efekt: po pierwsze, zagnieżdżony znacznik automatycznie powiększy obrazek po najechaniu nań kursorem myszy, po drugie zaś — zmniejsza (kurczy) tenże obrazek, kiedy kursor zostanie znad niego odsunięty. Te efekty są stosowane w ramach zdarzeń myszy kontrolki `Image` w `albumImage`.

System Flex udostępnia też efekty typowych animacji, w rodzaju przejść, wykręcania obrazków i modulowania kanałów alfa. Poza efektami wbudowanymi programiści mają do dyspozycji interfejs programistyczny do samodzielnego rysowania elementów, który

pozwala na zmontowanie nawet zaawansowanych animacji. Dalsze zagłębianie się w ten temat wymagałoby sięgnięcia do zagadnień projektowania grafiki i animacji, a to wykracza poza tematykę niniejszego podrozdziału.

Co do stylów, Flex obsługuje standardowe arkusze CSS. Z plikiem MXML można skojarzyć CSS, a wtedy kontrolki Flex będą odrysowywane zgodnie z narzuconymi stylami. Na przykład arkusz *songStyles.css* zawiera następującą deklarację CSS:

```
//:! gui/flex/songStyles.css
.headerText {
  font-family: Arial, "_sans";
  font-size: 16;
  font-weight: bold;
}

.boldText {
  font-family: Arial, "_sans";
  font-size: 11;
  font-weight: bold;
}
//:~
```

Plik ten jest importowany do aplikacji biblioteki utworów za pośrednictwem znacznika `Song` pliku MXML. Po zaimportowaniu arkusza stylów jego deklarację można stosować odnośnie kontrolek Flex z pliku MXML. W ramach przykładu deklaracja `boldText` arkusza jest wykorzystywana w definicji kontrolki pola tekstowego `TextArea` o identyfikatorze `songInfo`.

Zdarzenia

Interfejs użytkownika to automat stanów; realizuje akcje w ramach przejść pomiędzy ustalonymi stanami. W systemie Flex zmiany te są zarządzane za pośrednictwem systemu zdarzeń. Biblioteka klas Flex zawiera szeroki wachlarz kontrolek z wyczerpującymi zestawami zdarzeń pokrywających wszelkie aspekty poruszania myszą i obsługi klawiatury.

Na przykład atrybut `click` kontrolki `Button` reprezentuje jedno ze zdarzeń dostępnych dla tej kontrolki. Wartość skojarzona z atrybutem `click` może być funkcją albo osadzonym fragmentem skryptu. Przykładowo w pliku MXML kontener `ControlBar` przechowuje przycisk `refreshSongsButton` służący do odświeżania listy utworów. Z treści znacznika wiadomo, że kliknięcie tego przycisku wywoła zdarzenie `click`, które spowoduje wywołanie `songService.getSongs()`. W tym przykładzie zdarzenie `click` kontrolki `Button` odnosi się do `RemoteObject` reprezentującego zewnętrzną metodę Javy.

Połączenie z Javą

Znacznik `RemoteObject` umieszczony pod koniec pliku MXML ustanawia połączenie z zewnętrzną klasą Javy, konkretnie `gui.flex.SongService`. Klient Flex użyje metody `getSongs()` z klasy Java dla pozyskania danych do wypełnienia kontrolki `DataGrid`. W tym celu metoda ta musi być dostępna jako *usługa* — końcówka, z którą klient Flex wymienia komunikaty. Usługa definiowana znacznikiem `RemoteObject` posiada atrybut `source`, opisujący zewnętrzną klasę Javy i określający funkcję zwrotną ActionScript `onSongs()`,

która zostanie wywołana w reakcji na powrót z metody Javy. Zagnieżdżony znacznik method deklaruje metodę getSongs(), dzięki czemu zewnętrzna metoda Javy jest dostępna dla reszty aplikacji Flex.

Wszelkie wywołania usług w systemie Flex są wywołaniami asynchronicznymi, z odbiorem wyników za pośrednictwem wspomnianych funkcji zwrotnych. RemoteObject ponadto wywołuje okienko alarmowe w przypadku ewentualnego błędu.

Metoda getSongs() może teraz być wywoływana z aplikacji Flash za pomocą kodu ActionScript:

```
songService.getSongs();
```

Z racji konfiguracji pliku MXML spowoduje to wywołanie metody getSongs() klasy SongService:

```
//: gui/flex/SongService.java
package gui.flex;
import java.util.*;

public class SongService {
  private List<Song> songs = new ArrayList<Song>();
  public SongService() { fillTestData(); }
  public List<Song> getSongs() { return songs; }
  public void addSong(Song song) { songs.add(song); }
  public void removeSong(Song song) { songs.remove(song); }
  private void fillTestData() {
    addSong(new Song("Chocolate", "Snow Patrol",
      "Final Straw", "sp-final-straw.jpg",
      "chocolate.mp3"));
    addSong(new Song("Concerto No. 2 in E", "Hilary Hahn",
      "Bach: Violin Concertos", "hahn.jpg",
      "bachviolin2.mp3"));
    addSong(new Song("'Round Midnight", "Wes Montgomery",
      "The Artistry of Wes Montgomery",
      "wesmontgomery.jpg", "roundmidnight.mp3"));
  }
} ///:~
```

Każdy obiekt Song jest najzwyklejszym kontenerem danych:

```
//: gui/flex/Song.java
package gui.flex;

public class Song implements java.io.Serializable {
  private String name;
  private String artist;
  private String album;
  private String albumImageUrl;
  private String songMediaUrl;
  public Song() {}
  public Song(String name, String artist, String album,
  String albumImageUrl, String songMediaUrl) {
    this.name = name;
    this.artist = artist;
    this.album = album;
```

```
    this.albumImageUrl = albumImageUrl;
    this.songMediaUrl = songMediaUrl;
  }
  public void setAlbum(String album) { this.album = album;}
  public String getAlbum() { return album; }
  public void setAlbumImageUrl(String albumImageUrl) {
    this.albumImageUrl = albumImageUrl;
  }
  public String getAlbumImageUrl() { return albumImageUrl;}
  public void setArtist(String artist) {
    this.artist = artist;
  }
  public String getArtist() { return artist; }
  public void setName(String name) { this.name = name; }
  public String getName() { return name; }
  public void setSongMediaUrl(String songMediaUrl) {
    this.songMediaUrl = songMediaUrl;
  }
  public String getSongMediaUrl() { return songMediaUrl; }
} ///:~
```

Przy inicjalizacji aplikacji albo po naciśnięciu przycisku `refreshSongsButton` dochodzi do wywołania metody `getSongs()`, a po powrocie z niej wywoływana jest funkcja ActionScript `onSongs(event.result)`, wypełniająca kontrolkę `songGrid` danymi.

Oto listing skryptu ActionScript przeznaczonego dla kontrolki `Script` w pliku MXML:

```
//: gui/flex/songScript.as
function getSongs() {
  songService.getSongs();
}

function selectSong(event) {
  var song = songGrid.getItemAt(event.itemIndex);
  showSongInfo(song);
}

function showSongInfo(song) {
  songInfo.text = song.name + newline;
  songInfo.text += song.artist + newline;
  songInfo.text += song.album + newline;
  albumImage.source = song.albumImageUrl;
  songPlayer.contentPath = song.songMediaUrl;
  songPlayer.visible = true;
}

function onSongs(songs) {
  songGrid.dataProvider = songs;
} ///:~
```

Obsługę wyboru komórki kontrolki `DataGrid` definiuje atrybut zdarzenia `cellPress` w deklaracji kontrolki w pliki MXML:

```
cellPress="selectSong(event)"
```

Kiedy użytkownik kliknie w kontrolce `DataGrid` pozycję reprezentującą utwór, spowoduje wywołanie metody `selectSong()` z powyższego skryptu ActionScript.

Modele danych i wiązanie danych

Kontrolki mogą wprost odwoływać się do usług, a wywołania zwrotne zdarzeń skryptu ActionScript dają możliwość programowego aktualizowania kontrolek wizualnych w reakcji na zwrot danych przez usługi. Skrypt aktualizujący kontrolki jest zupełnie prosty, ale bywa rozwlekły i przez to nieporęczny, a sama operacja aktualizacji jest na tyle powszechna, że doczekała się automatyzacji ze strony systemu Flex za pomocą mechanizmu wiązania danych.

W najprostszym wydaniu wiązanie danych pozwala kontrolkom odwoływać się do danych wprost, bez pośrednictwa kodu kopiującego dane do kontrolki. W momencie aktualizacji danych kontrolka odwołująca się do tych danych jest również automatycznie aktualizowana, bez potrzeby interwencji programistycznej. Infrastruktura systemu Flex prawidłowo reaguje na zdarzenia zmiany danych, aktualizując wszelkie kontrolki dowiązane do tych danych.

Oto prosty przykład składni wiązania danych:

```
<mx:Slider id="mySlider"/>
<mx:Text text="{mySlider.value}">
```

W celu związania kontrolki z danymi należy skorzystać z odwołania w nawiasach klamrowych ({}). To, co znajdzie się pomiędzy tymi klamrami, będzie traktowane jako wyrażenie przeznaczone do ewaluacji przez system Flex.

W naszym przykładzie wartość pierwszej kontrolki, będącej suwakiem (`Slider`), jest wyświetlana w drugiej kontrolce — najzwyklejszym polu tekstowym (`Text`). W miarę zmian wartości suwaka następuje automatyczna aktualizacja właściwości `text` pola tekstowego. W ten sposób unika się programowego, jawnego odczytywania wartości suwaka i ustawiania wartości pola tekstowego.

Niektóre kontrolki, jak `Tree` czy `DataGrid` z naszej aplikacji przykładowej, są znacznie bardziej skomplikowane. Posiadają one właściwość `dataprovider` określającą dostarczyciela danych, to znaczy ustanawiającą wiązanie z kolekcjami danych. Funkcja skryptu ActionScript `onSongs()` pokazuje sposób związania metody `SongService.getSongs()` z atrybutem `dataprovider` kontrolki `DataGrid`. Zgodnie z deklaracją znacznika `RemoteObject` z pliku MXML funkcja ta stanowi wywołanie zwrotne, którym ActionScript obsługuje powrót z metody zewnętrznej.

W bardziej rozbudowanych aplikacjach, z bardziej skomplikowanymi modelami danych, na przykład w aplikacji korporacyjnej korzystającej z technologii DTO (Data Transfer Objects) albo aplikacji wymiany komunikatów z formatem danych zgodnym z rozbudowanymi schematami, trzeba uciec się do dalszej separacji kontrolek od źródeł danych. W programowaniu z użyciem systemu Flex owo rozprzężenie osiąga się przez deklarowanie obiektu „modelu" będącego ogólnym kontenerem danych MXML. Model nie zawiera żadnej logiki. Odzwierciedla on mechanizm DTO znajdujący zastosowanie w programowaniu korporacyjnym i strukturach innych języków programowania. Możemy więc łączyć kontrolki z modelem, a sam model wiązać z wejściami i wyjściami usług. W ten sposób dokonujemy rozdziału źródeł danych, usług i wizualnych konsumentów danych, wcielając w życie wzorzec projektowy *Model-View-Controller* (MVC). W rozleglej-

szych, bardziej rozbudowanych aplikacjach komplikacje związane z wdrożeniem modelu są niewielką ceną za elastyczność wynikającą z separacji charakterystycznej dla wzorca MVC.

Aplikacja Flex może korzystać nie tylko z zewnętrznych klas Javy, ale też z usług WWW opartych na SOAP i usług RESTful — służą do tego (odpowiednio) kontrolki `WebService` i `HttpService`. Dostęp do wszelkich usług podlega ograniczeniom wynikającym z zabezpieczeń autoryzacyjnych.

Kompilowanie i instalacja

W poprzednich przykładach kompilacja plików MXML mogła się odbywać bez określania opcji `-flexlib`, ale aby skompilować ostatni przykład, trzeba wskazać kompilatorowi położenie pliku *flex-config.xml* właśnie za pośrednictwem opcji `-flexlib`. W mojej instalacji zadziałało poniższe polecenie, w systemie Czytelnika być może trzeba będzie zmodyfikować ścieżkę dostępu zgodnie z zastaną konfiguracją (całe poniższe polecenie, choć złamane na wiele wierszy, stanowi w istocie pojedynczy wiersz polecenia):

```
//: gui/flex/build-command.txt
mxmlc -flexlib C:/"Program Files"/Macromedia/Flex/jrun4/servers/default/flex/WEB-
INF/flex songs.mxml
//:~
```

Powyższe polecenie skompiluje aplikację do postaci pliku SWF, nadającego się do odtworzenia w oknie przeglądarki WWW; pamiętaj jednak, że pakiet kodu źródłowego rozprowadzany wraz z książką nie zawiera plików MP3 i plików obrazków, do których odwołuje się aplikacja, trudno więc oczekiwać słyszalnych efektów odtwarzania utworów; sam szkielet aplikacji powinien jednak działać bez problemu.

Do tego należy skonfigurować serwer tak, aby mógł się skutecznie komunikować z plikami Javy wchodzącymi w skład aplikacji Flex. Pakiet próbny Flex zawiera serwer JRun, który po zainstalowaniu pakietu można uruchomić za pośrednictwem menu *Start* albo poniższego polecenia:

```
jrun -start default
```

Fakt pomyślnego uruchomienia serwera można sprawdzić, uruchamiając przeglądarkę i kierując ją pod adres *http://localhost:8700/samples* oraz uruchamiając znajdujące się tam aplikacje przykładowe (to zresztą świetny sposób na bliższe poznanie platformy Flex).

Zamiast kompilować aplikację z poziomu wiersza poleceń, można zdać się na pośrednictwo serwera. W tym celu należy umieścić pliki źródłowe aplikacji, arkusz stylów CSS i tak dalej w katalogu *jrun4/servers/default/flex*, a następnie odwołać się do aplikacji za pośrednictwem przeglądarki, wstukując adres *http://localhost:8700/flex/songs.mxml*.

Aby skutecznie uruchomić aplikację, należy skonfigurować stronę Javy i stronę systemu Flex.

Po stronie Javy: umieścić w katalogu *WEB-INF/classes* skompilowane pliki klas *Song.java* i *SongService.java*. To ten katalog, w którym umieszcza się pakiety WAR dla platformy J2EE. Alternatywnie, można oba pliki umieścić w archiwum JAR i umieścić całość

w *WEB-INF/lib*. Musi to być katalog odpowiadający strukturze pakietu Javy. Jeśli korzystasz z serwera JRun, powinieneś otrzymać pliki *jrun4/servers/default/flex/WEB-INF/ classes/gui/flex/Song.class* i *jrun4/servers/default/flex/WEB-INF/classes/gui/flex/SongService.class*. Potrzebne też będą pliki obrazków (JPG) i pliki MP3 wykorzystywane w aplikacji WWW (w przypadku serwera JRun katalogiem głównym aplikacji jest *jrun4/servers/default/flex*).

Po stronie systemu Flex: ze względów bezpieczeństwa Flex nie może odwoływać się do obiektów Javy bez specjalnego zezwolenia wyrażanego w pliku *flex-config.xml*. W przypadku serwera JRun plik ten znajduje się w katalogu *jrun4/servers/default/flex/WEB-INF/flex/flex-config.xml*. W pliku należy odnaleźć element `<remote-objects>` i przyjrzeć się liście `<whitelist>`; widnieje tam poniższa notka:

```
<!--
For security, the whitelist is locked down by default.
Uncomment the source element below to enable access to all classes
during development.

We strongly recommend not allowing access to all source files
in production, since this exposes Java and Flex system classes.
<source>*</source>
-->
```

Aby zezwolić na dostęp, należy tam usunąć znaki komentarza dla elementu `<source>`, tak aby widniał jako element `<source>*</source>`. Ciekawych znaczenia tego i pozostałych wpisów pliku konfiguracyjnego odsyłam do dokumentacji systemu Flex.

Ćwiczenie 38. Skompiluj „prosty przykład składni wiązania z danymi" z tego podrozdziału (3).

Ćwiczenie 39. Pakiet kodu źródłowego rozprowadzany wraz z książką nie zawiera plików MP3 ani obrazków JPG wykorzystywanych w *SongService.java*. Zdobądź jakieś utwory MP3 oraz obrazki JPG i zmodyfikuj klasę SongService.java tak, aby korzystała z ich nazw plików; pobierz próbny pakiet Flex skompiluj, zainstaluj i uruchom aplikację (4).

Aplikacje SWT

W bibliotece Swing przyjęto model rysowania elementów interfejsu piksel po pikselu, w sposób zupełnie oderwany od platformowych procedur odrysowywania interfejsu. Swing jest więc biblioteką zupełnie niezależną od istniejącej w danym systemie biblioteki elementów interfejsu użytkownika. Co innego w SWT, gdzie wypośrodkowano pomiędzy wykorzystywaniem gotowych komponentów systemu operacyjnego a syntetyzowaniem brakujących. W efekcie powstaje aplikacja bardzo upodobniona do tego, do czego przyzwyczaił się użytkownik danego systemu, za to z niekiedy wyraźnym przyspieszeniem interfejsu w porównaniu do biblioteki Swing. Do tego programowanie interfejsu z użyciem biblioteki SWT jest prostsze niż w Swingu, gdzie interfejs może stać się znaczącą częścią aplikacji[13].

[13] W uzyskiwaniu informacji o SWT i przygotowaniu przykładów bardzo pomógł mi Chris Grindstaff.

Ponieważ SWT wykorzystuje zastaną infrastrukturę systemu operacyjnego, automatycznie czerpie z zalet tej infrastruktury i jej cech niekoniecznie dostępnych w bibliotece Swing — przykładem może być choćby funkcja renderowania subpikselowego w Windows polepszająca wygląd czcionek na ekranach LCD.

Podrozdział ten nie ma być wyczerpującym omówieniem biblioteki SWT; ma jedynie dać przedsmak jej stosowania i porównać ją do Swing. Sam się przekonasz o liczbie rozmaitych elementów interfejsu SWT i o prostocie ich użycia. Zainteresowanych szczegółami już na wstępie odsyłam do dokumentacji z licznymi przykładami, dostępnej w witrynie *www.eclipse.org*. Na rynku wydawniczym dostępne są też książki traktujące o programowaniu z użyciem SWT.

Instalowanie SWT

Aplikacje SWT wymagają pobrania i zainstalowania biblioteki SWT, rozprowadzanej w ramach projektu Eclipse. Należy w tym celu udać się pod adres *www.eclipse.org/downloads/* i wybrać odpowiedni dla swojej lokalizacji serwer lustrzany. Z niego należy pobrać aktualną dystrybucję Eclipse i zlokalizować w niej skompresowany plik z nazwą zaczynającą się od ciągu „swt" i zawierającą określenie platformy systemowej, (np. „win32"). W pliku tym znajduje się archiwum *swt.jar*. Najprostszym sposobem zainstalowania swt.jar jest umieszczenie tego pliku w katalogu *jre/lib/ext* (dzięki temu nie trzeba będzie wprowadzać żadnych zmian do ścieżki `CLASSPATH`). Po rozpakowaniu biblioteki SWT znajdziesz dodatkowe pliki, które trzeba zainstalować w odpowiednich lokalizacjach danego systemu. Na przykład pakiet dla platformy Win32 zawiera pliki DLL, które należy umieścić w obrębie ścieżki `java.library.path` (zwykle pokrywa się ona ze ścieżkami dostępu zmiennej środowiskowej `PATH`; bieżącą wartość `java.library.path` można podejrzeć, uruchamiając program *object/ShowProperties.java*). Po tych czynnościach powinieneś móc w sposób transparentny kompilować i uruchamiać aplikacje SWT tak jak wszystkie inne programy Javy.

Dokumentacja SWT jest dystrybuowana w osobnym pakiecie.

Sposób alternatywny polega na zainstalowaniu środowiska programistycznego Eclipse, które obejmuje zarówno bibliotekę SWT, jak i dokumentację SWT, w formacie nadającym się do przeglądania za pomocą wbudowanego systemu pomocy.

Ahoj, SWT

Zacznijmy od najprostszej możliwej aplikacji, która — tradycyjnie — powita się z użytkownikiem i światem:

```
// : swt/HelloSWT.java
// {Requires: org.eclipse.swt.widgets.Display; Trzeba zainstalować
// bibliotekę SWT spod adresu http://www.eclipse.org }
import org.eclipse.swt.widgets.*;

public class HelloSWT {
  public static void main(String [] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
```

```
    shell.setText("Ahoj, SWT!"); // Pasek tytułowy
    shell.open();
    while(!shell.isDisposed())
      if(!display.readAndDispatch())
        display.sleep();
    display.dispose();
  }
} ///:~
```

Jeśli pobrałeś pakiet kodu źródłowego towarzyszący książce, odkryjesz, że dyrektywa `Requires` ląduje w pliku sterującym kompilacją *build.xml* systemu Ant w roli wymogów wstępnych dla kompilacji przykładów z katalogu *swt*; wszystkie pliki importujące pakiet `org.eclipse.swt` wymagają do kompilacji zainstalowania biblioteki SWT ze strony *www.eclipse.org*.

Klasa `Display` zarządza połączeniem pomiędzy SWT a danym systemem operacyjnym — to element *mostu* pomiędzy systemem operacyjnym a biblioteką SWT. Z kolei `Shell` to główne okno aplikacji, w którym osadzane są wszystkie pozostałe komponenty. Wywołanie metody `setText()` powoduje ustawienie tekstu na pasku tytułowym okna zgodnie z wartością argumentu.

Aby wyświetlić okno (a więc aplikację jako taką), należy na rzecz obiektu klasy `Shell` wywołać metodę `open()`.

Biblioteka Swing ukrywa pętlę obsługi zdarzeń przed programistą, tymczasem SWT wymusza jej jawne zapisanie. Na szczycie pętli sprawdzamy, czy okno główne zostało zniszczone — tutaj mamy szansę wykonać kod sprzątający po aplikacji. Oznacza to jednak, że wątek metody `main()` jest równocześnie wątkiem obsługi interfejsu użytkownika. W Swingu w tle tworzony był osobny wątek rozprowadzający i obsługujący zdarzenia związane interfejsem — w SWT całość obsługi interfejsu użytkownika odbywa się w wątku metody `main()`. Domyślna obecność tylko jednego wątku zmniejsza ryzyko zaciemnienia interfejsu wątkami.

Zauważ, że dzięki temu nie musisz troszczyć się o przekazywanie zadań do interfejsu użytkownika, jak to miało miejsce w bibliotece Swing. SWT zajmuje się tym automatycznie, a w przypadku próby manipulowania elementem interfejsu z poziomu złego wątku zgłosi wyjątek. Jedynie przy rozwidlaniu wątków programu w celu wykonywania długotrwałych zadań, należy (jak w Swingu) zadbać o przekazywanie informacji o zmianach do wątku interfejsu. W SWT służą do tego trzy metody, które wywołuje się na rzecz obiektu `Display`: `asyncExec(Runnable)`, `syncExec(Runnable)` i `timerExec(int, Runnable)`.

Aktywność wątku metody `main()` skupia się na wywołaniu metody `readAndDispatch()` obiektu klasy `Display` (implikuje to obecność pojedynczego obiektu `Display` dla każdej aplikacji SWT). Metoda `readAndDispatch()` zwraca wartość `true`, jeśli w kolejce zdarzeń znajdują się dalsze zdarzenia oczekujące na obsługę. W takim przypadku należy niezwłocznie wywołać tę metodę ponownie. Jeśli kolejka zdarzeń jest pusta, to przed następnym sprawdzaniem stanu kolejki zdarzeń należy obiekt `Display` na jakiś czas uśpić wywołaniem metody `sleep()`.

Po zakończeniu programu należy jawnie wywołać `dispose()` na rzecz obiektu `Display`. SWT często wymaga jawnego zwalniania zasobów, bo są to zwykle zasoby czerpane z systemu operacyjnego, które w wyniku zawłaszczania mogłyby ulec wyczerpaniu.

Aby dowieść, że faktyczne okno główne programu reprezentuje obiekt `Shell`, wystarczy w programie utworzyć większą liczbę takich obiektów:

```
//: swt/ShellsAreMainWindows.java
import org.eclipse.swt.widgets.*;

public class ShellsAreMainWindows {
  static Shell[] shells = new Shell[10];
  public static void main(String [] args) {
    Display display = new Display();
    for(int i = 0; i < shells.length; i++) {
      shells[i] = new Shell(display);
      shells[i].setText("Shell nr " + i);
      shells[i].open();
    }
    while(!shellsDisposed())
      if(!display.readAndDispatch())
        display.sleep();
    display.dispose();
  }
  static boolean shellsDisposed() {
    for(int i = 0; i < shells.length; i++)
      if(shells[i].isDisposed())
        return true;
    return false;
  }
} ///:~
```

Uruchomienie programu utworzy dziesięć okien głównych. Z racji struktury programu zamknięcie któregokolwiek z tych okien spowoduje zamknięcie również wszystkich pozostałych.

Biblioteka SWT korzysta z własnych menedżerów układu — odmiennych niż w bibliotece Swing, ale bazujących na identycznych koncepcjach. Oto nieco bardziej rozbudowany przykład, który pobiera tekst z wywołania `System.getProperties()` i umieszcza go w oknie programu:

```
//: swt/DisplayProperties.java
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import java.io.*;

public class DisplayProperties {
  public static void main(String [] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Okno właściwości");
    shell.setLayout(new FillLayout());
    Text text = new Text(shell, SWT.WRAP | SWT.V_SCROLL);
    StringWriter props = new StringWriter();
    System.getProperties().list(new PrintWriter(props));
    text.setText(props.toString());
    shell.open();
    while(!shell.isDisposed())
      if(!display.readAndDispatch())
```

```
        display.sleep();
      display.dispose();
    }
  } ///:~
```

W SWT wszystkie elementy interfejsu (ang. *widgets*) muszą posiadać element nadrzędny podtypu `Composite`; obiekt taki musi być przekazany w postaci pierwszego argumentu konstruktora każdego elementu. Widać to choćby w konstruktorze pola tekstowego `Text`: pierwszym argumentem wywołania konstruktora jest obiekt `shell`. Praktycznie wszystkie konstruktory SWT przyjmują również argument pozwalający na określenie dyrektyw stylu, zależnych od możliwości danego elementu interfejsu. Dyrektywy można łączyć, składając odpowiadające im stałe w sumę bitową, jak w powyższym przykładzie.

Przy konfigurowaniu obiektu `Text` przekazałem do niego znaczniki stylu wymuszające zawijanie tekstu i automatycznie dodające do pola tekstowego pionowy pasek przewijania (w razie potrzeby). Przekonasz się niebawem, że cała biblioteka SWT jest silnie ukierunkowana na konstruktory — wiele atrybutów elementów interfejsu da się ustawić wyłącznie z poziomu konstruktora. Listę znaczników dyrektyw stylu należy dla każdego elementu sprawdzać w jego dokumentacji. Niektóre konstruktory przyjmują argumenty znacznika nawet wtedy, kiedy dokumentacja nie wymienia żadnych wartości; pozwala to na przyszłe rozbudowywanie elementu bez zmieniania jego interfejsu.

Eliminowanie powtarzającego się kodu

Zanim przejdziemy dalej, chciałbym zaznaczyć, że w każdej aplikacji SWT wykonuje się pewien zestaw typowych operacji, zupełnie tak jak w Swingu, gdzie analogiczne operacje udało się nam wyodrębnić do klasy `SwingConsole`. W przypadku SWT chodzi o tworzenie obiektu `Display`, pozyskanie z niego obiektu `Shell`, zainicjowanie pętli `readAndDispatch()` i tym podobne. Oczywiście w przypadkach specjalnych można z tych czynności zrezygnować, ale są one na tyle typowe, że warto pokusić się o stworzenie szkieletu eliminującego powtarzający się kod na wzór `net.mindview.util.SwingConsole`.

W tym celu zmusimy wszystkie aplikacje do implementowania wspólnego interfejsu:

```
//: swt/util/SWTApplication.java
package swt.util;
import org.eclipse.swt.widgets.*;

public interface SWTApplication {
  void createContents(Composite parent);
} ///:~
```

Aplikacja otrzymuje obiekt `Composite` (`Shell` jest podklasą `Composite`) i powinna go wykorzystać do utworzenia całości interfejsu w ramach implementacji metody `createContents()`. Metoda `SWTConsole.run()` w odpowiednim momencie wywoła metodę `createContents()`, ustawiając rozmiar okna zgodnie z wartościami przekazanymi przez użytkownika w wywołaniu `run()`, otworzy okno i uruchomi pętlę zdarzeń; wreszcie przy wychodzeniu z programu zamknie okno:

```
//: swt/util/SWTConsole.java
package swt.util;
import org.eclipse.swt.widgets.*;
```

```
public class SWTConsole {
  public static void
  run(SWTApplication swtApp, int width, int height) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText(swtApp.getClass().getSimpleName());
    swtApp.createContents(shell);
    shell.setSize(width, height);
    shell.open();
    while(!shell.isDisposed()) {
      if(!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
} ///:~
```

Powyższy kod dodatkowo ustawia treść paska tytułowego zgodnie z nazwą klasy `SWTApplication` oraz szerokość i wysokość okna głównego.

Spróbujmy wykorzystać szkielet `SWTConsole` do napisania odmiany programu *DisplayProperties.java*, wypisującej zmienne środowiskowe systemu lokalnego:

```
//: swt/DisplayEnvironment.java
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import java.util.*;

public class DisplayEnvironment implements SWTApplication {
  public void createContents(Composite parent) {
    parent.setLayout(new FillLayout());
    Text text = new Text(parent, SWT.WRAP | SWT.V_SCROLL);
    for(Map.Entry entry: System.getenv().entrySet()) {
      text.append(entry.getKey() + ": " +
        entry.getValue() + "\n");
    }
  }
  public static void main(String [] args) {
    SWTConsole.run(new DisplayEnvironment(), 800, 600);
  }
} ///:~
```

`SWTConsole` pozwala skupić się na faktycznej zawartości aplikacji i nie zawracać sobie głowy powtarzającymi się wciąż drobiazgami.

Ćwiczenie 40. Zmień program *DisplayProperties.java* tak, aby korzystał z klasy `SWTConsole` (4).

Ćwiczenie 41. Zmień program *DisplayEnvironment.java* tak, aby nie korzystał z klasy `SWTConsole` (4).

Menu

W ramach ilustracji podstaw tworzenia menu poniższy program wczyta własny kod źródłowy i podzieli go na słowa, wypełniając nimi menu programu:

```
//: swt/Menus.java
// Zabawa z menu.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import java.util.*;
import net.mindview.util.*;

public class Menus implements SWTApplication {
  private static Shell shell;
  public void createContents(Composite parent) {
    shell = parent.getShell();
    Menu bar = new Menu(shell, SWT.BAR);
    shell.setMenuBar(bar);
    Set<String> words = new TreeSet<String>(
      new TextFile("Menus.java", "\\W+"));
    Iterator<String> it = words.iterator();
    while(it.next().matches("[0-9]+"))
      ; // Ominięcie liczb.
    MenuItem[] mItem = new MenuItem[7];
    for(int i = 0; i < mItem.length; i++) {
      mItem[i] = new MenuItem(bar, SWT.CASCADE);
      mItem[i].setText(it.next());
      Menu submenu = new Menu(shell, SWT.DROP_DOWN);
      mItem[i].setMenu(submenu);
    }
    int i = 0;
    while(it.hasNext()) {
      addItem(bar, it, mItem[i]);
      i = (i + 1) % mItem.length;
    }
  }
  static Listener listener = new Listener() {
    public void handleEvent(Event e) {
      System.out.println(e.toString());
    }
  };
  void
  addItem(Menu bar, Iterator<String> it, MenuItem mItem) {
    MenuItem item = new MenuItem(mItem.getMenu().SWT.PUSH);
    item.addListener(SWT.Selection, listener);
    item.setText(it.next());
  }
  public static void main(String[] args) {
    SWTConsole.run(new Menus(), 600, 200);
  }
} ///:~
```

Menu musi być umieszczone w oknie (Shell), a klasa Composite pozwala na pozyskanie obiektu okna głównego za pośrednictwem metody getShell(). Klasa TextFile, opisywana już w książce, została zapożyczona z biblioteki net.mindview.util; program wy-

pełnia kontener TreeSet słowami, przy okazji porządkując je alfabetycznie. W wyniku porządkowania na początku kontenera lądują cyfry i liczby, które ignorujemy. Na podstawie strumienia słów tworzymy menu najwyższego poziomu, potem poszczególne menu rozwijane — aż do wyczerpania słów.

W reakcji na wybranie którejś z pozycji menu odbiornik Listener najzwyczajniej wypisuje informacje o zdarzeniu — można się przekonać, jakie dane niesie ze sobą zdarzenie. Po uruchomieniu programu przekonasz się, że w skład tych informacji wchodzi etykieta menu, co pozwala na obsługę wyboru menu na podstawie nazwy pozycji; alternatywą jest utworzenie osobnych odbiorników dla poszczególnych pozycji (to sposób bezpieczniejszy, ze względu na ewentualną internacjonalizację menu).

Panele zakładek, przyciski i zdarzenia

SWT dysponuje bogatym zestawem kontrolek zwanych tu *widgetami*. Kontrolki podstawowe zostały zebrane w pakiecie org.eclipse.swt.widgets; pakiet org.eclipse.swt.custom zawiera kontrolki bardziej wyszukane. Polecam dokumentację obu pakietów.

Aby zademonstrować wygląd i zachowanie podstawowych kontrolek, poniższy przykład rozmieści szereg podprzykładów na zakładkach okna głównego. Przy okazji poznasz sposób tworzenia obiektów Composite (analogicznych do obiektów JPanel) w celu osadzania jednych elementów interfejsu w innych:

```
//: swt/TabbedPane.java
// Komponenty SWT na zakładkach.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.browser.*;

public class TabbedPane implements SWTApplication {
  private static TabFolder folder;
  private static Shell shell;
  public void createContents(Composite parent) {
    shell = parent.getShell();
    parent.setLayout(new FillLayout());
    folder = new TabFolder(shell, SWT.BORDER);
    labelTab();
    directoryDialogTab();
    buttonTab();
    sliderTab();
    scribbleTab();
    browserTab();
  }
  public static void labelTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Etykieta"); // Tekst na zakładce
    tab.setToolTipText("Prosta etykieta tekstowa");
    Label label = new Label(folder, SWT.CENTER);
    label.setText("Tekst etykiety");
    tab.setControl(label);
```

```
  }
  public static void directoryDialogTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Okno dialogowe wyboru katalogu");
    tab.setToolTipText("Wybierz katalog");
    final Button b = new Button(folder, SWT.PUSH);
    b.setText("Wybierz katalog");
    b.addListener(SWT.MouseDown, new Listener() {
        public void handleEvent(Event e) {
          DirectoryDialog dd = new DirectoryDialog(shell);
          String path = dd.open();
          if(path != null)
            b.setText(path);
        }
      });
    tab.setControl(b);
  }
  public static void buttonTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Przyciski");
    tab.setToolTipText("Różne rodzaje przycisków");
    Composite composite = new Composite(folder, SWT.NONE);
    composite.setLayout(new GridLayout(4, true));
    for(int dir : new int[]{
        SWT.UP, SWT.RIGHT, SWT.LEFT, SWT.DOWN
      }) {
      Button b = new Button(composite, SWT.ARROW | dir);
      b.addListener(SWT.MouseDown, listener);
    }
    newButton(composite, SWT.CHECK, "Pole wyboru");
    newButton(composite, SWT.PUSH, "Przycisk zwykły");
    newButton(composite, SWT.RADIO, "Przycisk wyboru");
    newButton(composite, SWT.TOGGLE, "Przełącznik");
    newButton(composite, SWT.FLAT, "Przycisk płaski");
    tab.setControl(composite);
  }
  private static Listener listener = new Listener() {
      public void handleEvent(Event e) {
        MessageBox m = new MessageBox(shell, SWT.OK);
        m.setMessage(e.toString());
        m.open();
      }
    };
  private static void newButton(Composite composite,
    int type, String label) {
    Button b = new Button(composite, type);
    b.setText(label);
    b.addListener(SWT.MouseDown, listener);
  }
  public static void sliderTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Suwaki i wskaźniki postępu");
    tab.setToolTipText("Suwak powiązany ze wskaźnikiem postępu");
    Composite composite = new Composite(folder, SWT.NONE);
    composite.setLayout(new GridLayout(2, true));
    final Slider slider =
      new Slider(composite, SWT.HORIZONTAL);
```

```
    final ProgressBar progress =
      new ProgressBar(composite, SWT.HORIZONTAL);
    slider.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent event) {
        progress.setSelection(slider.getSelection());
      }
    });
    tab.setControl(composite);
  }
  public static void scribbleTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Szkicownik");
    tab.setToolTipText("Prosta grafika: rysowanie");
    final Canvas canvas = new Canvas(folder, SWT.NONE);
    ScribbleMouseListener sml= new ScribbleMouseListener();
    canvas.addMouseListener(sml);
    canvas.addMouseMoveListener(sml);
    tab.setControl(canvas);
  }
  private static class ScribbleMouseListener
    extends MouseAdapter implements MouseMoveListener {
    private Point p = new Point(0, 0);
    public void mouseMove(MouseEvent e) {
      if((e.stateMask & SWT.BUTTON1) == 0)
        return;
      GC gc = new GC((Canvas)e.widget);
      gc.drawLine(p.x, p.y, e.x, e.y);
      gc.dispose();
      updatePoint(e);
    }
    public void mouseDown(MouseEvent e) { updatePoint(e); }
    private void updatePoint(MouseEvent e) {
      p.x = e.x;
      p.y = e.y;
    }
  }
  public static void browserTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText("Przeglądarka");
    tab.setToolTipText("Przeglądarka WWW");
    Browser browser = null;
    try {
      browser = new Browser(folder, SWT.NONE);
    } catch(SWTError e) {
      Label label = new Label(folder, SWT.BORDER);
      label.setText("Nie można zainicjalizować przeglądarki");
      tab.setControl(label);
    }
    if(browser != null) {
      browser.setUrl("http://www.mindview.net");
      tab.setControl(browser);
    }
  }
  public static void main(String[] args) {
    SWTConsole.run(new TabbedPane(), 800, 600);
  }
} ///:~
```

Metoda `createContents()` konfiguruje układ ogólny panelu, a następnie wywołuje metody wypełniające poszczególne zakładki. Tekst wyświetlany na każdej z zakładek jest ustawiany wywołaniem `setText()` (na zakładce można też tworzyć przyciski i grafikę); każda jest też opatrywana podpowiedzią wyskakującą. Na końcu każdej metody następuje wywołanie `setControl()`, które dodaje kontrolkę tworzoną w obrębie danej metody do przestrzeni danej zakładki.

Metoda `labelTab()` tworzy zakładkę z najzwyczajniejszą etykietą tekstową. `directoryDialogTab()` umieszcza na swojej zakładce standardowe okno dialogowe wyboru katalogu; nazwa wybranego katalogu jest przepisywana na przycisk.

Zakładka przycisków, konstruowana w metodzie `buttonTab()`, zawiera podstawowe odmiany przycisków SWT. Zakładka `sliderTab()` to powtórzenie przykładu z biblioteki Swing, z powiązaniem suwaka ze wskaźnikiem postępu.

Zakładka tworzona w metodzie `scribbleTab()` to przykład tworzenia własnych grafiki. Program rysujący składa się tu z zaledwie kilku wierszy kodu.

Wreszcie zakładka wynikowa metody `browserTab()` pokazuje siłę komponentu `Browser` z biblioteki SWT — w pełni funkcjonalnej przeglądarki WWW zamkniętej w pojedynczym komponencie.

Grafika

Oto program *SineWave.java* w wersji dla SWT:

```
//: swt/SineWave.java
// Program SineWave.java w wersji dla SWT.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;

class SineDraw extends Canvas {
  private static final int SCALEFACTOR = 200;
  private int cycles;
  private int points;
  private double[] sines;
  private int[] pts;
  public SineDraw(Composite parent, int style) {
    super(parent, style);
    addPaintListener(new PaintListener() {
      public void paintControl(PaintEvent e) {
        int maxWidth = getSize().x;
        double hstep = (double)maxWidth / (double)points;
        int maxHeight = getSize().y;
        pts = new int[points];
        for(int i = 0; i < points; i++)
          pts[i] = (int)((sines[i] * maxHeight / 2 * .95)
            + (maxHeight / 2));
        e.gc.setForeground(
          e.display.getSystemColor(SWT.COLOR_RED));
```

```
        for(int i = 1; i < points; i++) {
          int x1 = (int)((i - 1) * hstep);
          int x2 = (int)(i * hstep);
          int y1 = pts[i - 1];
          int y2 = pts[i];
          e.gc.drawLine(x1, y1, x2, y2);
        }
      }
    });
    setCycles(5);
  }
  public void setCycles(int newCycles) {
    cycles = newCycles;
    points = SCALEFACTOR * cycles * 2;
    sines = new double[points];
    for(int i = 0; i < points; i++) {
      double radians = (Math.PI / SCALEFACTOR) * i;
      sines[i] = Math.sin(radians);
    }
    redraw();
  }
}

public class SineWave implements SWTApplication {
  private SineDraw sines;
  private Slider slider;
  public void createContents(Composite parent) {
    parent.setLayout(new GridLayout(1, true));
    sines = new SineDraw(parent, SWT.NONE);
    sines.setLayoutData(
      new GridData(SWT.FILL, SWT.FILL, true, true));
    sines.setFocus();
    slider = new Slider(parent, SWT.HORIZONTAL);
    slider.setValues(5, 1, 30, 1, 1, 1);
    slider.setLayoutData(
      new GridData(SWT.FILL, SWT.DEFAULT, true, false));
    slider.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent event) {
        sines.setCycles(slider.getSelection());
      }
    });
  }
  public static void main(String[] args) {
    SWTConsole.run(new SineWave(), 700, 400);
  }
} ///:~
```

Powierzchnią do rysowania własnej grafiki jest tu nie `JPanel`, ale obiekt `Canvas`.

Porównanie obu wersji programu ujawnia, że sama klasa `SineDraw` jest w obu przypadkach niemal identyczna. W SWT kontekst graficzny `gc` pobiera się z obiektu zdarzenia przekazywanego do odbiornika `PaintListener`, w Swingu jego rolę pełnił obiekt `Graphics` przekazywany wprost do metody `paintComponent()`, ale czynności wykonywane na obiekcie kontekstu graficznego są bardzo podobne, a metoda `setCycles()` wprost identyczna.

Metoda createContents() wymaga nieco więcej kodu niż w wersji dla biblioteki Swing, w celu ułożenia elementów i połączenia suwaka z jego odbiornikiem, lecz podstawowe czynności programu są w zasadzie identyczne.

Współbieżność w SWT

Same AWT/Swing to biblioteki jednowątkowe, ale ową jednowątkowość można łatwo zaburzyć tak, żeby uzyskać program niedeterministyczny. Zasadniczo należałoby unikać angażowania wielu wątków do obsługi interfejsu, bo konkurujące wątki mogą ze sobą kolidować.

W SWT nie ma takiej możliwości — próba odwołania się do interfejsu z poziomu wątku innego niż wątek metody main() spowoduje wyjątek. Powstrzymuje to niedoświadczonych programistów przed pomyłkami tego rodzaju, które owocują trudnymi do odszukania i zdiagnozowania błędami programu.

Oto tłumaczenie programu *ColorBoxes.java* dla biblioteki SWT:

```
//: swt/ColorBoxes.java
// Wersja programu ColorBoxes.java dla biblioteki SWT.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

class CBox extends Canvas implements Runnable {
  class CBoxPaintListener implements PaintListener {
    public void paintControl(PaintEvent e) {
      Color color = new Color(e.display, cColor);
      e.gc.setBackground(color);
      Point size = getSize();
      e.gc.fillRectangle(0, 0, size.x, size.y);
      color.dispose();
    }
  }
  private static Random rand = new Random();
  private static RGB newColor() {
    return new RGB(rand.nextInt(255),
      rand.nextInt(255), rand.nextInt(255));
  }
  private int pause;
  private RGB cColor = newColor();
  public CBox(Composite parent, int pause) {
    super(parent, SWT.NONE);
    this.pause = pause;
    addPaintListener(new CBoxPaintListener());
  }
  public void run() {
    try {
      while(!Thread.interrupted()) {
```

```
            cColor = newColor();
          getDisplay().asyncExec(new Runnable() {
            public void run() {
              try { redraw(); } catch(SWTException e) {}
              // Wyjątek SWTException jest normalny, jeśli obiekt
              // nadrzędny zostaje przerwany z poziomu potomnego.
            }
          });
          TimeUnit.MILLISECONDS.sleep(pause);
        }
      } catch(InterruptedException e) {
        // Dopuszczalny sposób wyjścia
      } catch(SWTException e) {
        // Dopuszczalny sposób wyjścia: nasz rodzic został
        // przerwany przez nas.
      }
    }
  }

  public class ColorBoxes implements SWTApplication {
    private int grid = 12;
    private int pause = 50;
    public void createContents(Composite parent) {
      GridLayout gridLayout = new GridLayout(grid, true);
      gridLayout.horizontalSpacing = 0;
      gridLayout.verticalSpacing = 0;
      parent.setLayout(gridLayout);
      ExecutorService exec = new DaemonThreadPoolExecutor();
      for(int i = 0; i < (grid * grid); i++) {
        final CBox cb = new CBox(parent, pause);
        cb.setLayoutData(new GridData(GridData.FILL_BOTH));
        exec.execute(cb);
      }
    }
    public static void main(String[] args) {
      ColorBoxes boxes = new ColorBoxes();
      if(args.length > 0)
        boxes.grid = new Integer(args[0]);
      if(args.length > 1)
        boxes.pause = new Integer(args[1]);
      SWTConsole.run(boxes, 500, 400);
    }
  } ///:~
```

Rysowanie, jak w poprzednim przykładzie, jest kontrolowane za pośrednictwem odbiornika `PaintListener` z metodą `paintControl()`, wywoływaną, kiedy wątek SWT jest gotowy do odrysowania danego komponentu. Odbiornik `PaintListener` jest rejestrowany w konstruktorze `CBox`.

W tej wersji `CBox` zdecydowaną odmianę stanowi metoda `run()`, która nie może zwyczajnie i wprost wywołać metody `redraw()`, ale musi przekazać `redraw()` do metody `asyncExec()` obiektu `Display`, co jest odpowiednikiem wywołania `SwingUtilities.invokeLater()`. Zastąpienie tego wywołania bezpośrednim wywołaniem metody `redraw()` spowoduje przerwanie programu.

W trakcie działania programu można zaobserwować drobne nieścisłości graficzne — niekiedy dają się zauważyć poziome linie przebiegające przez obszar odrysowywania. To dlatego, że domyślnie interfejs SWT nie podlega podwójnemu buforowaniu, jak to ma miejsce w bibliotece Swing. Spróbuj uruchomić obok siebie wersje SWT i Swing, a różnica stanie się całkiem wyraźna. Nic nie stoi na przeszkodzie, aby samodzielnie oprogramować podwójne buforowanie w SWT; przykłady można znaleźć na stronie *www.eclipse.org*.

Ćwiczenie 42. Zmodyfikuj program *swt/ColorBoxes.java* tak, aby rozsiewał w obszarze okna punkty („gwiazdki") i potem losowo zmieniał kolory „gwiazdek" (4).

SWT czy Swing?

Tak skrócone wprowadzenie utrudnia uzyskanie pełnego obrazu, ale wiemy już co nieco o SWT i widać z tego, że w wielu sytuacjach programowanie z użyciem biblioteki SWT jest prostsze niż z użyciem biblioteki Swing. Mimo to oprogramowanie interfejsu SWT również może okazać się skomplikowane, więc nie to powinno być głównym motywem wybierania tej, a nie innej biblioteki GUI. Ważniejsze, że SWT daje użytkownikowi większy komfort korzystania z aplikacji przez upodobnianie jej elementów do pozostałych aplikacji działających na danej platformie. Ważna jest też zwiększona reaktywność interfejsu SWT. Tam, gdzie obie te cechy mają znaczenie drugorzędne, polecałbym stosowanie biblioteki Swing.

Ćwiczenie 43. Wybierz jeden z przykładów dotyczących biblioteki Swing, który nie został przystosowany do SWT, i przetłumacz przykład pod kątem biblioteki SWT (uwaga: to dobre ćwiczenie na zadanie domowe — jego rozwiązania nie ma w zestawie rozwiązań rozprowadzanych wraz z książką) (6).

Podsumowanie

Spośród wszystkich bibliotek Javy biblioteka GUI najbardziej zmieniła się w Java 2. Biblioteka AWT Java 1.0 była otwarcie krytykowana jako jeden z najgorszych projektów w historii i mimo tego, że pozwalała tworzyć przenośne programy, to ich interfejs „był tak samo marny na wszystkich platformach". Był również o wiele bardziej ograniczony i nie tak ładny, jak interfejs normalnych aplikacji dostępnych dla poszczególnych platform.

Wszystko się zmieniło, kiedy w Java 1.1 wprowadzono nowy model zdarzeń oraz technologię JavaBeans — stało się możliwe tworzenie komponentów GUI, które można było łatwo wykorzystać w wizualnych środowiskach do tworzenia aplikacji nawet poprzez przeciąganie i upuszczanie na formatce. Nowy projekt modelu zdarzeń i komponentów JavaBean wyraźnie pokazuje, jak silny nacisk położono na ułatwienie programowania i poprawę możliwości pielęgnacji kodu (coś, co nie było tak oczywiste w AWT 1.0). Nie można było uznać prac za zakończone przed pojawieniem się nowych komponentów GUI — klas JFC/Swing. Programowanie wieloplatformowego interfejsu użytkownika dzięki wykorzystaniu komponentów Swing może być w pełni cywilizowanym doświadczeniem.

Właściwie brakowało jedynie dobrego narzędzia do tworzenia aplikacji i na tym polu zachodzi obecnie prawdziwa rewolucja. Jeśli chcemy, by narzędzia programistyczne języka zamkniętego stawały się lepsze, to musimy trzymać kciuki i mieć nadzieję, że producenci dadzą nam to, czego chcemy, ale Java jest środowiskiem otwartym, które nie tylko pozwala na współistnienie konkurencyjnych narzędzi programistycznych, ale wręcz do nich zachęca. Jeśli natomiast jakieś narzędzie ma zostać potraktowane poważnie, musi być przystosowane do JavaBeans. Oznacza to wyrównane szanse dla każdego: jeżeli pojawi się lepsze narzędzie programistyczne, nie jesteśmy w żaden sposób związani z aktualnie używanym — można bez problemu przenieść się do nowszego i zwiększyć swoją produktywność. Nigdy wcześniej nie pojawił się tego typu, nastawiony na współzawodnictwo rynek środowisk programistycznych. Może to prowadzić jedynie do pozytywnych efektów, sprzyjających poprawie warunków pracy programisty.

Rozdział miał na celu jedynie przybliżenie potęgi programowania interfejsów użytkownika na tyle, by umożliwić rozpoczęcie pracy z nimi i pokazać, jak łatwo można poruszać się po bibliotekach. To, co przedstawiłem do tej pory, prawdopodobnie wystarczy do wykonania większości projektowanych interfejsów użytkownika. Jednak Swing, SWT i Flex (Flash) potrafią o wiele więcej. Prawdopodobnie można w nich znaleźć rozwiązania wszystkiego, co tylko można sobie wyobrazić.

Zasoby

Pod adresem *www.galbraiths.org/presentations* można znaleźć niezłe omówienie bibliotek Swing i SWT w postaci prezentacji Bena Galbraitha.

Rozwiązania wybranych zadań można znaleźć w elektronicznym dokumencie *The Thinking in Java Annotated Solution Guide*, dostępnym za niewielką opłatą pod adresem *www.MindView.net*.

Dodatek A

Materiały uzupełniające

Uzupełnieniem informacji zawartych w niniejszej książce są suplementy, które można zamówić w witrynie MindView, usługi dostępne za jej pośrednictwem oraz seminaria.

Wszelkie dostępne materiały uzupełniające zostały opisane w niniejszym dodatku, żebyś mógł ocenić, czy któreś z nich Ci się przydadzą.

Suplementy do pobrania

Przede wszystkim z witryny *www.MindView.net* można pobrać oryginalny kod źródłowy wszystkich przykładów prezentowanych w książce; pakiet obejmuje pliki dla systemu automatycznej kompilacji Ant oraz inne pliki pomocnicze niezbędne do skutecznej kompilacji i uruchomienia wszystkich przykładów z książki.

Do tego niektóre części samej książki zyskały w nowym wydaniu postać osobnych, elektronicznych suplementów obejmujących zagadnienia:

- klonowania obiektów,
- przekazywania i zwracania obiektów,
- analizy i projektowania,
- fragmenty rozdziałów trzeciego wydania *Thinking in Java*, które nie trafiły do druku w wydaniu czwartym.

Thinking in C

Na stronie *www.MindView.net* znajdziesz udostępnianą nicodpłatnie prezentację szkoleniową *Thinking in C*. Ma ona postać prezentacji w formacie Flash, przygotowanej przez Chucka Allisona dla firmy MindView. Stanowi ona kurs wprowadzający do składni języka C, jego operatorów i funkcji, na których z kolei bazuje składnia Javy.

Do odtworzenia prezentacji wymagane jest zainstalowanie w systemie odtwarzacza Flash Player, dostępnego do pobrania ze strony *www.Macromedia.com*.

...enie Thinking in Java

Moja firma, MindView, Inc., prowadzi pięciodniowe, praktyczne szkolenia przygotowawcze, oparte na materiale zamieszczonym w niniejszej książce. Są to zarówno szkolenia otwarte, jak i prowadzone w miejscu pracy. Wcześniej nosiły nazwę *Hands-On Java*. W założeniu są to szkolenia wprowadzające, stanowiące podstawę do bardziej zaawansowanych kursów. Składają się z wybranego materiału z poszczególnych rozdziałów, a po zakończeniu lekcji przeprowadzane są monitorowane ćwiczenia, podczas których każdemu z uczestników poświęcamy nieco czasu. Informacje dotyczące terminarza szkoleń, miejsc ich przeprowadzania, zaświadczeń oraz wszelkich innych szczegółów można znaleźć w witrynie *www.MindView.net*.

Szkolenie na CD-ROM-ie Hands-On Java

Zawiera ono poszerzony materiał szkolenia *Thinking in Java* i także bazuje na niniejszej książce. Zapewnia osiągnięcie takiego samego poziomu wiedzy jak po „normalnym" szkoleniu, bez konieczności jakichkolwiek wyjazdów i ponoszenia dodatkowych kosztów. Kurs składa się z wykładów (odtwarzanych dźwiękowo) oraz sekwencji slajdów odpowiadających poszczególnym rozdziałom książki. Stworzyłem je osobiście i występuję także jako lektor. Szkolenie można uruchomić na każdym komputerze wyposażonym w odtwarzacz Flash Player. CD-ROM ze szkoleniem można zamówić w witrynie *www.MindView.net*, gdzie można zapoznać się również z demonstracyjnymi wersjami próbnymi.

Szkolenie Thinking in Objects

To szkolenie wprowadzające do zagadnień programowania obiektowego z punktu widzenia projektanta. Szkolenie koncentruje się na procesie pisania i tworzenia systemów, a w szczególności na tzw. „zwinnych metodach" oraz „prostych metodologiach" (zwłaszcza na „programowaniu ekstremalnym" — XP). Przedstawimy ogólne wprowadzenie do metodologii, podstawowe narzędzia, takie jak technika planowania „karty indeksu" opisana w książce *Planning Extreme Programming* (Beck i Fowler, 2002), karty CRC do projektowania obiektów, programowanie w parach, planowanie iteracji, testowanie jednostkowe, automatyczną kompilację kodu, kontrola kodu źródłowego oraz inne podobne zagadnienia. W skład szkolenia wchodzi także projekt realizowany zgodnie z założeniami programowania ekstremalnego (i wykonywany etapami przez cały tydzień).

Jeśli przymierzasz się do rozpoczęcia projektu z użyciem technik projektowania obiektowego, możemy wykorzystać Twoje zadanie jako przykład w trakcie kursu i pod koniec tygodnia ustalić pierwszą wersję projektu.

Informacje dotyczące terminarza szkoleń, miejsc ich przeprowadzania, zaświadczeń oraz wszelkich innych szczegółów można znaleźć w witrynie *www.MindView.net*.

Thinking in Enterprise Java

Książka powstała przez połączenie wybranych rozdziałów wchodzących wcześniej w skład książki *Thinking in Java*, poświęconych bardziej złożonym zagadnieniom. Nie stanowi ona bynajmniej drugiej części *Thinking in Java*, lecz koncentruje się na przedstawieniu zaawansowanych zagadnień związanych z tworzeniem aplikacji korporacyjnych. Książkę można (w takiej czy innej postaci, bo wciąż ewoluuje) znaleźć i pobrać z witryny *www.MindView.net*. Ponieważ to osobna publikacja, będzie z pewnością uzupełniana o kolejne zagadnienia. Ostatecznie ma zaś stanowić proste i zrozumiałe wprowadzeniem do podstawowych technologii wykorzystywanych przy tworzeniu aplikacji korporacyjnych, dzięki któremu będziesz przygotowany do skorzystania z bardziej zaawansowanych materiałów dotyczących tego zagadnienia.

W książce zostaną poruszone między innymi następujące tematy (nie jest to lista kompletna):

- wprowadzenie do zagadnień programowania korporacyjnego;
- programowanie sieciowe przy wykorzystaniu gniazd i kanałów;
- Remote Method Invocation (RMI);
- nawiązywanie połączeń z bazami danych;
- usługi nazewnicze i katalogowe;
- serwlety;
- JavaServer Pages;
- znaczniki, JSP Fragments oraz język wyrażeń;
- automatyzacja tworzenia interfejsu użytkownika;
- Enterprise JavaBeans;
- XML;
- usługi sieci Web (Web Services);
- zautomatyzowane testowanie.

Elektroniczną, bieżącą wersję książki można pobrać z witryny *www.MindView.net*.

,ıg in Patterns (with Java)

Jednym z najważniejszych czynników rozwoju projektowania zorientowanego obiektowo jest ruch rozwijający ideę „wzorców projektowych" opisany przez Gammę, Helma, Johnsona oraz Vlissidesa w książce *Design Patterns* (Addison-Wesley, 1995). Książka przedstawia 23 różne rozwiązania konkretnych klas problemów, zaimplementowane głównie w języku C++. *Design Patterns* stała się podstawą i źródłem słownictwa niemal obowiązkowo używanego przez wszystkich programistów korzystających z języków zorientowanych obiektowo. Książka *Thinking in Patterns* przedstawia podstawowe pojęcia związane ze wzorcami projektowymi oraz przykłady napisane w Javie. Nie została ona stworzona jako translacja książki *Design Patterns* wykorzystująca Javę jako język, w którym zaimplementowano przykłady, lecz raczej jako spojrzenie na wzorce projektowe pod kątem języka Java. Jej tematyka nie ogranicza się do oryginalnych 23 wzorców projektowych, lecz obejmuje także inne pomysły oraz techniki rozwiązywania problemów.

Zaczątkiem tej książki był rozdział z pierwszego wydania *Thinking in Java*. Niemniej jednak, wraz z rozwojem opisywanych w nim zagadnień, stało się oczywiste, że musi on przybrać postać niezależnej książki. W chwili gdy piszę ten dodatek, praca nad książką *Thinking in Patterns (with Java)* wciąż trwa, jednak prezentowany w niej materiał został opracowany i sprawdzony podczas wielu szkoleń *Objects & Patterns* (które aktualnie zostało rozbite na dwa odrębne szkolenia: *Thinking in Objects* oraz *Thinking in Patterns*).

Po więcej informacji o książce zapraszam na strony *www.MindView.net*.

Szkolenie Thinking in Patterns

Powstało na bazie popularnego szkolenia *Objects & Patterns*[1] prowadzonego od kilku lat przez Billa Vennersa oraz przeze mnie, którego materiał rozrósł się na tyle, iż zdecydowaliśmy się podzielić je na dwa kursy: *Thinking in Patterns* oraz *Thinking in Objects* (które opisałem wcześniej).

Szkolenie bardzo dokładnie odpowiada materiałowi książki *Thinking in Patterns*, a zatem najlepszym sposobem poznania jego tematyki jest pobranie elektronicznej wersji książki z witryny *www.MindView.net*.

Znaczną część prezentacji stanowi przykład procesu ewolucji projektowania, który rozpoczyna się od rozwiązania początkowego, a następnie przechodzi modyfikacje logiczne oraz proces ewolucji rozwiązania, aż do wykształcenia się bardziej poprawnego projektu. Ostatni z prezentowanych projektów (symulacja przetwarzania i odzyskiwania odpadów) przechodził długą ewolucję, można ją uznać za przykład ukazujący drogę, jaką mogą przejść Twoje własne rozwiązania — zaczynając od właściwego rozwiązania konkretnego problemu, które z czasem przekształca się w elastyczne rozwiązanie całej klasy problemów.

[1] Obiekty i wzorce.

- Powiększ elastyczność tworzonych projektów.
- Postaw na możliwości rozbudowy i wielokrotnego wykorzystania kodu.
- Stosując język wzorców projektowych, usprawnij wymienianie informacji na temat projektu.

Po każdym wykładzie uczestnicy szkolenia wykonują kilka ćwiczeń, podczas których pomagamy im wykorzystać konkretne wzorce projektowe do rozwiązania określonych problemów programistycznych.

Informacje dotyczące terminarza szkoleń, miejsc ich przeprowadzania, zaświadczeń oraz wszelkich innych szczegółów można znaleźć w witrynie *www.MindView.net*.

Konsultacja i analiza projektów

Moja firma zajmuje się także konsultowaniem i nadzorowaniem projektów, analizą projektów i implementacji, udziela również porad dotyczących procesu implementacji projektu — zwłaszcza pierwszego projektu realizowanego w Javie. Wszelkie informacje na temat dostępności usług tego typu można znaleźć w witrynie *www.MindView.net*.

Dodatek B

Zasoby

Oprogramowanie

JDK z java.sun.com. Nawet jeśli zdecydujesz się używać środowiska programowania innego producenta, dobrze jest mieć JDK pod ręką w razie pojawienia się np. błędu kompilatora. JDK służy jako sprawdzian — jeśli wystąpiłby w nim błąd, to prawdopodobnie byłby powszechnie znany.

Dokumentacja Javy w wersji HTML z java.sun.com. Nie znalazłem jeszcze książki z opisem standardowych bibliotek Javy, która nie byłaby przestarzała lub nie brakowałoby w niej informacji. Chociaż dokumentacja HTML Suna jest przekrojowa, zawiera drobne błędy i czasem jest lakoniczna, to jednak wszystkie klasy i metody są *tam* przynajmniej wymienione. Czasami można czuć się nieswojo, korzystając z dokumentacji w Internecie zamiast drukowanej książki, warto jednak przemóc się i najpierw zajrzeć do wersji HTML, aby zyskać przynajmniej ogólny obraz problemu. Jeżeli to nie pomoże, sięgaj po książkę.

Edytory i środowiska programistyczne

W tej dziedzinie obserwujemy wyjątkowo zdrową konkurencję. W znacznej mierze jest to oprogramowanie nicodpłatne (a w przypadku odpłatnego zazwyczaj dostępne są wersje próbne), najlepiej więc po prostu wypróbować wymienione programy i wybrać ten, który okaże się najlepiej dostosowany do konkretnych potrzeb i oczekiwań. Oto kilka propozycji:

JEdit — darmowy edytor autorstwa Slavy Pestova, napisany w języku Java, a więc pozwalający się przekonać, jak wyglądają aplikacje pisane w tym języku. Edytor oparty na systemie wtyczek powstających w łonie aktywnej społeczności użytkowników. Do pobrania spod adresu *www.jedit.org*.

NetBeans — darmowe zintegrowane środowisko programistyczne firmy Sun rozprowadzane za pośrednictwem witryny *www.netbeans.org*. Przygotowane do graficznego projektowania interfejsu użytkownika, wygodnej edycji kodu, diagnostyki itd.

Eclipse — otwarty (w sensie open-source) projekt wspierany między innymi przez firmę IBM. Platforma Eclipse została zaprojektowana do roli dającej się rozbudowywać podstawy, na bazie której można montować własne, samodzielne aplikacje. W ramach tego projektu powstała choćby biblioteka elementów interfejsu graficznego SWT omawiana w rozdziale „Graficzne interfejsy użytkownika". Do pobrania z witryny *www.Eclipse.org*.

IntelliJ IDEA — płatne zintegrowane środowisko programistyczne, cenione przez znaczne grono programistów języka Java, z których część uważa, że IDEA jest zawsze o krok, dwa przed Eclipse; zwiększona dynamika rozwoju wynika zapewne po części z tego, że IntelliJ skupia się na samym zintegrowanym środowisku programistycznym, nie aspirując do miana fundamentu dla oprogramowania (a więc nie zawierając własnych bibliotek podstawowych, jak SWT w Eclipse). Wersję próbną środowiska można pobrać ze strony *www.jetbrains.com*.

Książki

Core Java 2, Horstmann & Cornell, tom I — *Fundamentals* (Prentice-Hall, 1999) (*Java 2. Podstawy*, Helion, 2003), tom II — *Advanced Features*, 2000 (*Java 2. Techniki zaawansowane*, Helion, 2003). Obszerna i wyczerpująca — jest pierwszym źródłem, do którego sięgam w poszukiwaniu odpowiedzi. Polecam ją, kiedy skończysz czytać *Thinking in Java* i będziesz chciał wypłynąć na szersze wody.

The Java Class Libraries: An Annotated Reference, Patrick Chan i Rosanna Lee (Addison-Wesley, 1997). To, czym *powinna* być dokumentacja internetowa — wystarczająca liczba opisów, aby nadawała się do użytku. Jeden z recenzentów *Thinking in Java* powiedział: „Gdybym miał tylko jedną książkę o Javie, byłaby to właśnie ta (oczywiście oprócz Twojej)". Ja nie jestem nią tak zachwycony. Jest wielka, droga i nie zadowala mnie jakość przykładów. Jednak można do niej zajrzeć, gdy napotka się trudny do rozwiązania problem — zawiera dokładniejsze omówienia (idzie za tym zwiększona objętość) niż podobne książki.

Java Network Programming, wydanie drugie, Elliotte Rusty Harold (O'Reilly, 2000). Nie rozumiałem działania Javy w sieci, zanim nic odkryłem tej książki. Także strona internetowa autora, Café au Lait, przedstawia stymulujący i aktualny pogląd na tworzenie oprogramowania w Javie, nieobciążony zależnościami od jakichkolwiek producentów. Znajduje się tam także dział stale aktualizowanych wiadomości ze świata Javy. Patrz *www.cafeaulait.org*.

Design Patterns, Gamma, Helm, Johnson i Vlissides (Addison-Wesley, 1995). Przełomowa książka, która wprowadziła wzorce do programowania.

Practical Algorithms for Programmers, Binstock i Rex (Addison-Wesley, 1995). Algorytmy są napisane w C, całkiem łatwo można je więc przetłumaczyć na Javę. Każdy z nich jest wyczerpująco objaśniony.

Refactoring to Patterns, Joshua Kerievsky (Addison-Wesley, 2005) (*Refaktoryzacja do wzorców projektowych*, Helion, 2005). Łączy refaktoryzację z wzorcami projektowymi.

Największą zaletą książki jest uwidocznienie ewolucji, jaka zachodzi w projekcie w miarę rozpoznawania i wdrażania w nim kolejnych wzorców projektowych.

The Art of UNIX Programming, Eric Raymond (Addison-Wesley, 2004) (*UNIX. Sztuka programowania*, Helion, 2004). Choć Java to język wieloplatformowy, zadomowienie się Javy na serwerach zmusza programistów do zdobywania wiedzy o Uniksach i Linuksie. Książka Erica stanowi znakomite wprowadzenie do historii i filozofii tych systemów operacyjnych, a dla zainteresowanych korzeniami komputeryzacji stanowi po prostu świetną i wciągającą lekturę.

Analiza i projektowanie

Extreme Programming Explained, wydanie drugie, Kent Beck i Cynthia Andres (Addison-Wesley, 2005). Uwielbiam tę książkę. Owszem, stosuję dosyć radykalne podejście, ale zawsze przewidywałem, że istnieje inny, dużo lepszy sposób tworzenia programów, i uważam, że XP bardzo się do tego zbliża. Jedyną książką, która wywarła na mnie podobne wrażenie, jest *Peopleware* (opisana niżej), która dotyczy przede wszystkim środowiska i interakcji w zespołach ludzi. Autorzy *Extreme Programming Explained* piszą o programowaniu — i to o wszystkim, łącznie z najnowszymi „odkryciami". Posuwa się nawet do twierdzenia, że posługiwanie się rysunkami jest w porządku — pod warunkiem że nie spędzamy nad nimi za dużo czasu i za bardzo się do nich nie przywiązujemy (zauważysz, że ta książka nie ma na okładce „Pieczęci aprobaty UML"). Mógłbym podjąć decyzję, czy przyjąć pracę w firmie, zależnie od tego, czy używają XP. Niezbyt gruba książka z krótkimi rozdziałami, czyta się ją bez wysiłku, a rozmyśla o niej z podekscytowaniem. Zaczynasz sobie wyobrażać pracę w takiej atmosferze i pojawia się wizja całkiem nowego świata.

UML Distilled, wydanie drugie, Martin Fowler (Addison-Wesley, 2000). Pierwsze spotkanie z UML może być zniechęcające ze względu na ogromną ilość diagramów i szczegółów. Według Fowlera większość z tych szczegółów nie jest niezbędna, dzięki czemu autor może zajmować się najważniejszymi zagadnieniami. W przypadku większości projektów, konieczna jest znajomość jedynie kilku narzędzi tworzenia diagramów, a celem autora jest stworzenie dobrego projektu, a nie zwracanie uwagi na wykorzystywane akcesoria. Miła, cienka i czytelna książka; pierwsza po jaką należy sięgnąć, aby zrozumieć UML.

Domain Driven-Design, Eric Evans (Addison-Wesley, 2004). Książka koncentruje się na *modelu dziedzinowym* jako najważniejszym aspekcie procesu projektowania. To ciekawe i ważne przesunięcie nacisku w dziedzinie projektowania, pomocne projektantom w utrzymaniu odpowiedniego poziomu abstrakcji.

The Unified Software Development Process, Ivar Jacobsen, Grady Booch i James Rumbaugh (Addison-Wesley, 1999). Byłem z góry uprzedzony do tej książki, ponieważ wyglądała na kolejny nudny podręcznik. Zostałem pozytywnie zaskoczony — tylko niektóre fragmenty książki zawierają wyjaśnienia spraw, które nie były chyba do końca zrozumiałe dla samych autorów. Jednak całość jest nie tylko jasna, ale wręcz przyjemna w czytaniu. A co najważniejsze, zawiera dużo praktycznych przykładów. Nie jest to *Extreme Programming* (i nie ma jego jasności testów), ma jednak silny związek z UML — nawet jeśli nie można zastosować XP, wielu ludzi twierdzi, że „UML jest świetny" (niezależnie

od rzeczywistych z nim doświadczeń), i prawdopodobnie będą chcieli go wykorzystać. Ta książka powinna być okrętem flagowym UML i odsyłam do niej, jeśli po przeczytaniu *UML Distilled* Fowlera będziesz chciał poznać szczegóły.

Zanim wybierze się jakąkolwiek technikę, dobrze jest poznać punkt widzenia kogoś, kto żadnej z nich nie sprzedaje. Łatwo jest zastosować metodologię, nie rozumiejąc naprawdę, czego się od niej wymaga i co może ona zrobić. Wystarczającym powodem jest to, że inni jej używają. Jeżeli jednak ludzie uwierzą, że coś rozwiąże ich problemy, będą to stosować (to jest eksperymentowanie — i to jest pozytywne). Gdy jednak nie uda im się rozwiązać problemu, podwoją wysiłki i zaczną rozgłaszać, jaką wspaniałą rzecz odkryli (to jest zaprzeczanie — i to jest negatywne). Jeżeli inni wsiądą do Twojej łódki, nie będziesz już sam — nawet jeśli łódka płynie donikąd (lub powoli tonie).

Nie chcę sugerować, że wszystkie technologie prowadzą donikąd, ale powinieneś za wszelką cenę pozostać w fazie eksperymentowania („To nie działa, wypróbujmy coś innego"), a nie w fazie zaprzeczania („Nie, to nie jest problem. Wszystko jest cudownie, nie trzeba żadnych zmian"). Myślę, że następujące książki, przeczytane *przed* podjęciem decyzji, pozwolą Ci zachować właściwe podejście.

Software Creativity, Robert Glass (Prentice-Hall, 1995). Najlepsza książka omawiająca *perspektywy* metodologii, jaką czytałem. Jest to zbiór krótkich esejów i notatek, które Glass napisał lub czasem otrzymał od innych (jednym z autorów jest P.J. Plauger), odzwierciedlających jego długoletnią pracę i przemyślenia. Są zabawne i nie dłuższe niż potrzeba, żeby przekazać niezbędne treści; autor nie snuje dywagacji i nie nudzi. Nie rzuca także słów na wiatr; podaje setki odniesień do innych publikacji i książek. Każdy programista i kierownik powinien przeczytać tę książkę, zanim zacznie brodzić w „bagnie" metodologii.

Software Runaways: Monumental Software Disasters, Robert Glass (Prentice-Hall, 1997). Wspaniałe w tej książce jest to, że zwraca uwagę na fakt, o którym zwykle się nie mówi: wiele projektów nie tylko zawodzi, ale ponosi spektakularną porażkę. Większość z nas wciąż myśli: „To nie zdarzy się *mnie*" (albo „To nie zdarzy się *znowu*"), i myślę, że naraża nas to na straty. Jeżeli pamiętasz, że coś zawsze może pójść źle, będzie Ci łatwiej dopilnować, aby poszło dobrze.

Peopleware, wydanie drugie, Tom Demarco i Timothy Lister (Dorset House, 1999). To książka, którą *koniecznie* trzeba przeczytać. Jej lektura to doskonała zabawa, która dodatkowo jest w stanie zatrząść twym światem i zniszczyć przyjmowane wcześniej założenia. Autorzy znają się na tworzeniu oprogramowania, ta książka dotyczy jednak projektów i zespołów programistycznych. Skupia się przy tym na *ludziach* i ich potrzebach, a nie na technologiach i ich wymaganiach. Mówi o tworzeniu środowiska, w którym ludzie będą szczęśliwi i produktywni, a nie o ustalaniu zasad, które uczynią z nich odpowiednie elementy maszyny. To ostatnie podejście jest — jak myślę — spowodowane przez programistów, którzy uśmiechają się i potakują, gdy wprowadzana jest metoda XYZ, a potem po cichutku pracują tak, jak do tej pory.

Secrets of Consulting: A Guide to Giving & Getting Advice Successfully, Gerald M. Weinberg (Dorset House, 1985). Doskonała książka, jedna z mych ulubionych. Jest idealna dla osób starających się zostać konsultantami oraz tych, którzy pracują z konsultantami i chcą lepiej wykonywać swoje zadania. Rozdziały są krótkie, wypełnione opowiadaniami

i anegdotami uczącymi, w jaki sposób dojść do sedna sprawy, wkładając w to jak najmniej wysiłku. Warto także sięgnąć po książkę *More Secrets of Consulting* wydaną w 2002 roku bądź inną książkę tego autora.

Complexity, M. Mitchell Waldrop (Simon & Schuster, 1992). Jest to kronika spotkań naukowców różnych dyscyplin, zbierających się w Santa Fe, w Nowym Meksyku, aby przedyskutować problemy, z którymi ich gałęzie wiedzy sobie nie radzą (rynek giełdowy w ekonomii, formowanie się życia w biologii, dlaczego ludzie robią, to co robią, socjologia itd.). Przez skrzyżowanie fizyki, ekonomii, chemii, matematyki, informatyki, socjologii i innych wytwarza się rozwojowe, interdyscyplinarne podejście do takich zagadnień. Co jednak ważniejsze, pojawia się nowy sposób *myślenia* o tych ultrazłożonych problemach: odchodzi się od matematycznego determinizmu i iluzji, że wszystko można przewidzieć, rozwiązując odpowiednie równanie, w stronę *obserwacji* i poszukiwania wzorca, a następnie prób emulacji tego wzorca wszystkimi dostępnymi środkami (w książce jest np. opisane wyłanianie się idei algorytmów genetycznych). Ten sposób myślenia staje się użyteczny, pomagając dostrzec sposoby kierowania coraz bardziej złożonymi projektami.

Python

Learning Python, wydanie drugie, Mark Lutz i David Ascher (O'Reilly, 2003) (*Python. Wprowadzenie*, Helion, 2002). Przyjemne wprowadzenie do tego, co szybko stało się moim ulubionym językiem, wspaniałym uzupełnieniem Javy. Ta książka zawiera również wprowadzenie do języka Jython, który pozwala na łączenie Javy i Pythona w jednym programie (interpreter Jythona jest kompilowany do kodu bajtowego Javy, zatem nic trzeba niczego więcej, aby to osiągnąć). Taka unia języków oferuje wielkie możliwości.

Lista moich książek

Lista utworzona w kolejności publikacji. Nie wszystkie są dostępne.

Computer Interfacing with Pascal & C, (wydane własnym nakładem pod znakiem wydawnictwa Eisys, 1988. Dostępne jedynie przez *www.MindView.net)*. Wprowadzenie do elektroniki od czasów, kiedy królował jeszcze CP/M, a DOS był parweniuszem. Wykorzystywałem języki programowania wysokiego poziomu i port równoległy komputera do budowy rozmaitych projektów elektronicznych. Książka powstała na podstawie moich artykułów zamieszczanych w pierwszym i najlepszym czasopiśmie, dla jakiego pisałem — *Micro Cornucopia* (parafrazując Larry'ego O'Briena, długoletniego wydawcę *Software Development Magazine*: najlepsze czasopismo komputerowe, jakie kiedykolwiek się ukazywało — zamieścili nawet plany zbudowania robota w doniczce!). Niestety, Micro C poszło w zapomnienie na długo przedtem, zanim pojawił się Internet. Stworzenie tej książki było jednak nader satysfakcjonującym doświadczeniem wydawniczym.

Using C++ (Osborne/McGraw-Hill, 1989). Jedna z pierwszych książek na temat C++. Nakład wyczerpany, zastąpiona przez wydanie drugie o zmienionej nazwie: *C++ Inside & Out*.

C++ Inside & Out (Osborne/McGraw-Hill, 1993). Jak wspomniałem, drugie, poprawione wydanie *Using C++*. Opis C++ w tej książce jest dosyć dokładny, był to jednak rok 1992. W zamierzeniu książkę tę miało zastąpić *Thinking in C++*. Możesz dowiedzieć się więcej na ten temat (i ściągnąć kody źródłowe) z witryny *www.MindView.net*.

Thinking in C++, wydanie pierwsze (Prentice Hall, 1995).

Thinking in C++, wydanie drugie, tom I (Prentice Hall, 2000) (*Thinking in C++. Edycja polska*, Helion, 2002). Do ściągnięcia z *www.MindView.net*[1].

Thinking in C++, wydanie drugie, tom II (Prentice Hall, 2003), napisana wspólnie z Chuckiem Allisonem. Do ściągnięcia z *www.MindView.net*.

Black Belt C++, the Master's Collection. Pod redakcją Bruce'a Eckela (M&T Books, 1994). Nakład wyczerpany. Zbiór rozdziałów napisanych przez rozmaitych specjalistów od C++, opartych na ich wystąpieniach w ramach wątku C++ na konferencji Software Development Conference, której przewodniczyłem. Okładka tej książki skłoniła mnie do przejęcia kontroli nad projektami okładek moich przyszłych publikacji.

Thinking in Java, wydanie pierwsze (Prentice Hall, 1998). Pierwsza edycja tej książki otrzymała nagrody Software Development Magazine Productivity Award, Java Developer's Journal Editor's Choice Award oraz JavaWorld Reader's Choice Award dla najlepszej książki. Do ściągnięcia z *www.MindView.net*.

Thinking in Java, wydanie drugie (Prentice Hall, 2000) (*Thinking in Java. Edycja polska*. Helion, 2001). To wydanie książki wygrało nagrodę JavaWorld Editor's Choice Award. Do ściągnięcia z *www.MindView.net*.

Thinking in Java, wydanie trzecie (Prentice Hall, 2003) (*Thinking in Java. Edycja polska*. Helion, 2003). To wydanie książki zdobyło nagrodę Jolt Award magazynu *Software Development Magazine* w kategorii najlepsza książka roku oraz inne wyróżnienia, wymienione na okładce książki. Do ściągnięcia z *www.MindView.net*.

[1] Według mnie jest to jedna z najlepszych i najbardziej wyczerpujących książek omawiających ten język. Poziom szczegółowości jest praktycznie na poziomie *Języka C++* Stroustrupa, a sama książka jest napisana bardzo przystępnie i czyta się ją bardzo przyjemnie —*przyp. red.*

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

Ó

P

Q

R

S

Ś

T

U

V

W

X

Y

Z

Ź

Ż