

Jeśli chcesz poznać Javę, nie szukaj dalej – oto pierwsza książka techniczna z graficznym interfejsem użytkownika!

Wydanie II. Obejmuje Javę 5.0

Rusz głową! Java



Dowiedz się, jak wątki mogą poprawić Twój życie



Uniknij zawstydzających błędów obiektowych

Sprawdź swój umysł na 42 zagadkach



Pomóż mózgowi utrwalić zagadnienia związane z Java



Poigraj w bibliotece Javy



Twórz atrakcyjne i użyteczne interfejsy graficzne aplikacji

O'REILLY®

Kathy Sierra, Bert Bates

Helion

Twórcy serii Rusz głową!

Kathy Sierra



Bert Bates



Kathy interesowała się teorią nauczania od czasów, gdy pracowała jako projektantka gier (pisała gry dla takich firm jak Virgin, MGM oraz Amblin). Znaczną część postaci, w jakiej została wydana niniejsza książka, Kathy opracowała podczas prowadzenia kursu New Media Authoring zorganizowanego w ramach programu dodatkowego studiów nad rozrywką na Uniwersytecie Kalifornijskim. Ostatnio Kathy pracowała w firmie Sun Microsystems jako główny instruktor, ucząc prowadzących kursy Javy tego, jak należy nauczać najnowszych technologii tego języka, była także jednym z głównych twórców kilku egzaminów certyfikujących dla programistów Javy opracowanych przez firmę Sun. Wraz z Bertem Batesem aktywnie wykorzystywała pomysły zawarte w niniejszej książce podczas nauczania setek instruktorów, programistów, a nawet osób całkowicie niezwiązanych z programowaniem. Jest także założycielką jednej z największych witryn przeznaczonych dla społeczności programistów używających Javy — www.javaranch.com — oraz blogu *Creating Passionate Users*.

Kathy jest także współautorką książek *Head First Servlets. Edycja polska*, *Head First EJB* oraz *Wzorce projektowe. Rusz głową!*.

W wolnym czasie Kathy cieszy się swoim nowym koniem rasy islandzkiej, jeźdzeniem na nartach, bieganiem i prędkością światła.

kathy@wickedlysmart.com

Choć Kathy i Bert starają się odpowiadać na możliwie jak najwięcej listów, to jednak ich ilość oraz częste podróże autorów sprawiają, że jest to zadanie bardzo trudne. Dlatego najlepszym (i najszybszym) sposobem uzyskania pomocy technicznej związanej z tematami poruszonymi w tej książce jest poszukanie jej na bardzo aktywnych forach dla początkujących użytkowników Javy (na przykład na witrynie <http://forum.4programmers.net/>).

Bert zajmuje się tworzeniem i projektowaniem programów, jednak wieloletnia praca nad zagadnieniami sztucznej inteligencji sprawiła, że zainteresował się teorią nauczania i nauczaniem z wykorzystaniem nowych technologii. Odkąd pamięta, uczył swoich klientów programowania. Ostatnio był członkiem grupy opracowującej kilka egzaminów certyfikacyjnych języka Java w firmie Sun.

Pierwszą dekadę swojej programistycznej kariery Bert spędził, podróżując po świecie i pomagając rozmówcom radiowym, takim jak: Radio New Zealand, Weather Channel czy też Art & Entertainment Network (A & E). Jednym z jego ulubionych projektów było stworzenie pełnego systemu symulacyjnego kolej na potrzeby Union Pacific Railroad.

Bert jest zapalonym graczem w Go i już od bardzo dawna pracuje nad symulatorem tej gry. Dość dobrze gra na gitarze, a obecnie próbuje swych sił w grze na banjo. Oprócz tego lubi spędzać czas, jeżdżąc na nartach, biegając oraz trenując (lub będąc uczonym przez) swojego islandzkiego konia Andy.

Bert pisał książki wspólne z Kathy, a obecnie ciężko pracuje nad kolejną ich grupą (zajrzyj do bloga po najświezsze informacje na ten temat).

Mogą go czasem złapać na serwerze IGS GO (ma pseudonim *jackStraw*).

terrapin@wickedlysmart.com

Spis treści (skrócony)

Wprowadzenie	21
1. Szybki skok na głęboką wodę. <i>Przełamując zalew początkowych trudności</i>	33
2. Klasa i obiekty. <i>Wycieczka do Obiektowa</i>	59
3. Typy podstawowe i odwołania. <i>Poznaj swoje zmienne</i>	81
4. Metody wykorzystują składowe. <i>Jak działają obiekty?</i>	103
5. Pisanie programu. <i>Supermocne metody</i>	127
6. Poznaj Java API. <i>Korzystanie z biblioteki Javy</i>	155
7. Dziedziczenie i polimorfizm. <i>Wygodniejsze życie w Obiekcie</i>	193
8. Interfejsy i klasy abstrakcyjne. <i>Poważny polimorfizm</i>	225
9. Konstruktory i odśmiecacz. <i>Życie i śmierć obiektu</i>	263
10. Liczby oraz metody i składowe statyczne. <i>Liczby mają znaczenie</i>	301
11. Obsługa wyjątków. <i>Ryzykowne działania</i>	343
12. Tworzenie graficznego interfejsu użytkownika. <i>Historia bardzo graficzna</i>	379
13. Stosowanie biblioteki Swing. <i>Popracuj nad Swingiem</i>	425
14. Serializacja i operacje wejścia-wyjścia na plikach. <i>Zapisywanie obiektów</i>	453
15. Zagadnienia sieciowe i wątki. <i>Nawiąż połączenie</i>	495
16. Kolekcje i typy ogólne. <i>Struktury danych</i>	553
17. Pakiety, archiwa JAR i wdrażanie. <i>Rozpowszechnij swój kod</i>	605
18. Zdalne wdrażanie z użyciem RMI. <i>Przetwarzanie rozproszone</i>	629
A Ostatnie doprawianie kodu	671
B Dziesięć najważniejszych zagadnień, które niemal znalazły się w tej książce...	681

Spis treści (z prawdziwego zdarzenia)



Wprowadzenie

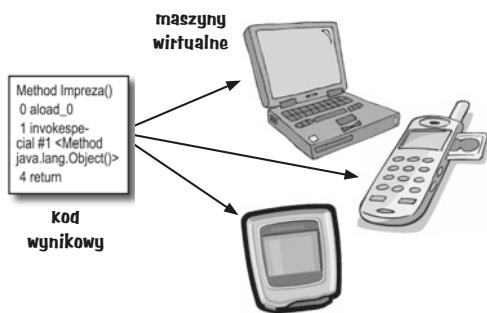
Twój mózg myśli o Javie. W tym rozdziale *Ty* próbujesz się czegoś dowiedzieć, natomiast Twój mózg robi Ci uprzejmość i stara się, aby te informacje nie zostały zapamiętane *na dłujo*. Myśli sobie: „Lepiej zostawić miejsce na ważniejsze rzeczy, takie jak dzikie zwierzęta, których należy się wystrzegać, lub rozważania, czy jeźdzenie na snowboardzie w stroju Adama to dobry pomysł”. A zatem jak można oszukać własny mózg i przekonać go, że od znajomości Javy zależy nasze życie?

Dla kogo jest przeznaczona ta książka?	22
Wiemy, co sobie myślisz	23
Metapoznanie — myślenie o myśleniu	25
Oto co możesz zrobić, aby zmusić swój mózg do posłuszeństwa	27
Czego potrzebujesz, aby skorzystać z tej książki?	28
Redaktorzy techniczni	30
Inne osoby, które można pochwalić	31

1

Przełamując zalew początkowych trudności

Java zabiera nas w nowe miejsca. Od momentu pojawienia się pierwszej, skromnej wersji o numerze 1.02 Java pociągała programistów ze względu na przyjazną składnię, cechy obiektowe, zarządzanie pamięcią, a przede wszystkim obietnicę przenośności. Od samego początku „wskoczymy na głęboką wodę” — napiszemy prosty program, skompilujemy go i wykonamy. Porozmawiamy o składni, pętlach, rozgałęzieniach oraz innych czynnikach, które sprawiają, że Java jest taka fajna. Zatem wskakuj!

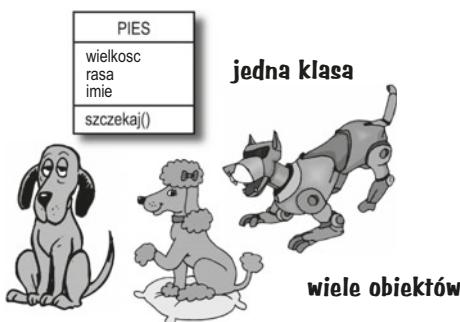


Jak działa Java?	34
Struktura kodu w Javie	39
Anatomia klas	40
Tworzenie klasy z metodą main	41
Pętle i pętle i...	43
Przykłady pętli while	44
Rozgałęzienia warunkowe	45
Tworzenie poważnej aplikacji biznesowej	46
Program krasomówczy	49

2

Wycieczka do Obiektowa

Mówiono mi, że będą tam same obiekty. W rozdziale 1. cały kod programu został umieszczony w metodzie `main()`. Nie jest to w pełni zgodne z zasadami programowania obiektowego. A zatem trzeba porzucić świat programowania proceduralnego i rozpocząć tworzenie własnych obiektów. Zobaczmy, co sprawia, że programowanie obiektowe w Javie jest takie fajne. Wyjaśnimy także, na czym polega różnica pomiędzy klasą a obiektem. W końcu pokażemy, w jaki sposób obiekty mogą ułatwić nam życie.



Wojna o fotel (albo Jak Obiekty Mogą Zmienić Twoje Życie)	60
Na plaży na laptopie Jurka	61
O tym beztrosko zapomniano napisać w specyfikacji	62
A co z metodą <code>obroc()</code> dla „sameby”?	64
Ta niepewność mnie zabije! Kto wygra Superfotel?	65
Tworzenie pierwszego obiektu	68
Tworzenie i testowanie obiektów Film	69
Szybko! Opuszczamy metodę <code>main!</code>	70

3

Poznaj swoje zmienne

Istnieją dwa rodzaje zmiennych: zmienne typów podstawowych oraz

odwołania. W programach będą istnieć także inne typy danych niż jedynie liczby całkowite,łańcuchy znaków i tablice. Co zrobić, jeśli chcielibyśmy stworzyć obiekt **WłaścicielZwierzaka** zawierający składową **Pies**? Albo **Pojazd** ze składową **Silnik**? W tym rozdziale ujawnimy tajemnice typów danych w Javie oraz pokażemy, co można zadeklarować jako zmienną, zapisać w zmiennej oraz co z taką zmienną można potem zrobić. W końcu przekonamy się, jak wygląda życie na automatycznie porządkowanej stercie.



Deklarowanie zmiennej	82
„Proszę podwójną. Albo nie — całkowitą!”	83
Naprawdę nie chcesz niczego rozsypywać	84
Tabela słów zarezerwowanych	85
Odwołanie do obiektu to jedynie inna wartość zmiennej	87
Życie na odśmiecanej stercie	89
Tablice także są obiektami	91
Tworzymy tablicę obiektów Pies	92
Przykładowy obiekt Pies	94

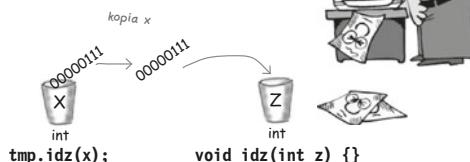
4

Jak działają obiekty?

Stan wpływa na działanie, a działanie wpływa na stan. Wiemy, że obiekty mają swój **stan** i określone **działanie** reprezentowane odpowiednio przez **składowe i metody**.

Teraz sprawdzimy, w jaki sposób stan i działanie obiektów są ze sobą *powiązane*. Otóż w swych działaniach obiekty wykorzystują swój unikalny stan. Innymi słowy, **metody wykorzystują wartości składowych obiektów**. Na przykład: „Jeśli pies waży mniej niż 20 kilogramów, szczekamy radośnie, w przeciwnym przypadku...”. **Spróbowajmy zmienić stan obiektu!**

Java przekazuje argumenty przez wartość.
To oznacza, że przekazywana jest kopia.



Wielkość ma wpływ na sposób szczenia	105
Do metod można przekazywać informacje	106
Metoda może coś zwrócić	107
Java przekazuje argumenty przez wartość	109
Ciekawe rozwiązania wykorzystujące parametry i wartości wynikowe	111
Hermetyzacja	112
Ukryj dane	113
Jak zachowują się obiekty w tablicy?	115
Deklarowanie i inicjalizacja składowych	116
Różnica pomiędzy składowymi a zmiennymi lokalnymi	117
Porównywanie zmiennych (typów podstawowych oraz odwołań)	118

5

Supermocne metody

Dodajmy naszym metodom nieco siły. Zajmowaliśmy się już zmiennymi, eksperymentowaliśmy z kilkoma obiektami i napisaliśmy kod paru programów. Jednak potrzeba nam więcej narzędzi. Na przykład **operatorów**. I **pętli**. Może przydałoby się **wygenerować kilka liczb losowych** i **zamienićłańcuch znaków na liczbę całkowitą**. O tak, to byłoby super. I dlaczego nie nauczyć się tego wszystkiego, tworząc coś rzeczywistego, by przekonać się, jak wygląda pisanie (i testowanie) programu w Javie od samego początku do końca. **Może jakasz grę**, taką jak „Zatopić portal” (podobną do gry w okrągły).

Stworzymy grę „Zatopić portal!”						
A	1	2	3	4	5	6
B						
C						
D			www.onet.pl			
E						
F						
G			www.wp.pl			

Napiszmy grę przypominającą „statki”, o nazwie „Zatopić portal”	128
Łagodne wprowadzenie do prostszej wersji gry	130
Pisanie implementacji metod	133
Pisanie kodu testowego dla klasy ProstyPortal	134
Kod testowy dla klasy ProstyPortal	135
Ostateczny kod klas ProstyPortal oraz ProstyPortalTester	138
Kod przygotowawczy klasy ProstyPortalGra	140
Trochę więcej o pętlach for	146
Różnica pomiędzy pętlami for i while	147
Rozszerzone pętle	148
Rzutowanie wartości typów podstawowych	149

6

Korzystanie z biblioteki Javy

Java jest wyposażona w setki gotowych klas. Jeśli tylko dowiesz się, jak znaleźć w bibliotece Javy (nazywanej także **Java API**) to, czego szukasz, nie będziesz musiał ponownie wymyślać koła. **W końcu masz ciekawsze rzeczy do roboty.** Jeśli masz zamiar napisać program, to równie dobrze możesz napisać tylko te jego fragmenty, które są unikalne. Standardowa biblioteka Javy to gigantyczny zbiór klas, które tylko czekają, aby użyć ich jako klocków przy tworzeniu programów.

„Dobrze jest wiedzieć, że w pakiecie `java.util` istnieje klasa `ArrayList`. Ale jak mogłabym się o tym sama dowiedzieć?”

— Julia, lat 31, modelka

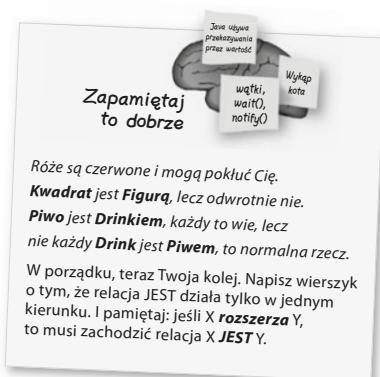


Ostatni rozdział zakończył się w dramatycznych okolicznościach — w programie znaleźliśmy błąd	156
Oryginalny kod przygotowawczy fragmentu metody sprawdz()	160
Niektóre możliwości klasy <code>ArrayList</code>	163
<code>ArrayList</code>	164
Porównanie klasy <code>ArrayList</code> ze zwyczajną tablicą	167
Napiszmy właściwą wersję gry „Zatopić portal”	170
Co (i kiedy) robią poszczególne obiekty w grze?	172
Kod przygotowawczy właściwej klasy <code>PortalGraMax</code>	174
Wyrażenia logiczne o bardzo dużych możliwościach	181
Stosowanie biblioteki (Java API)	184
Jak poznać API?	188

7

Wygodniejsze życie w Obiekcie

Planuj swoje programy, myśląc o przyszłości. A gdyby tak można tworzyć kod, który inni programiści mogliby rozbudowywać, i to w prosty sposób? A gdyby tak można było tworzyć elastyczny kod z myślą o tych najnowszych zmianach wprowadzanych w specyfikacji? Kiedy dowiesz się o Planie Polimorfizmu, poznasz 5 kroków służących projektowaniu lepszych klas, 3 sztuczki z polimorfizmem oraz 8 sposobów tworzenia elastycznego kodu, a jeśli zaczniesz już teraz, dodatkowo otrzymasz 4 podpowiedzi odnośnie do wykorzystywania dziedziczenia.



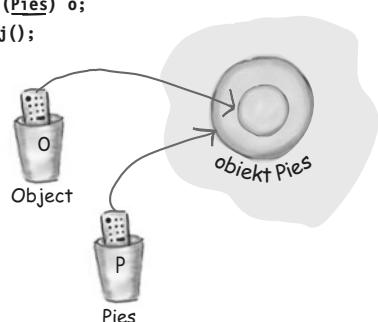
Zrozumienie dziedziczenia	196
Przykład dziedziczenia	197
Jakie metody należy przesłonić?	200
Jaka metoda jest wywoływana?	203
Projektowanie drzewa dziedziczenia	204
Ale poczekaj! To jeszcze nie wszystko!	206
Jak możesz określić, czy dobrze zaprojektowałeś hierarchię dziedziczenia?	207
Czy korzystanie z dziedziczenia przy projektowaniu klas jest „używaniem”, czy „nadużywaniem”?	209
Co w rzeczywistości daje nam dziedziczenie?	210
Jak dotrzymać kontraktu — reguły przesłaniania	218
Przeciążanie metody	219

8

Poważny polimorfizm

Dziedziczenie to tylko początek. Aby w pełni wykorzystać polimorfizm, niezbędne nam będą interfejsy. Musimy pójść dalej, poza proste dziedziczenie, i uzyskać elastyczność, jaką może zapewnić wyłącznie projektowanie i kodowanie z wykorzystaniem interfejsów. Czym jest interfejs? W 100% abstrakcyjną klasą. A czym jest klasa abstrakcyjna? To klasa, która nie pozwala na tworzenie obiektów. A co nam po takiej klasie? Przeczytaj, to się dowiesz...

```
Object o = lista.get(indeks);
Pies p = (Pies) o;
p.szczekaj();
```



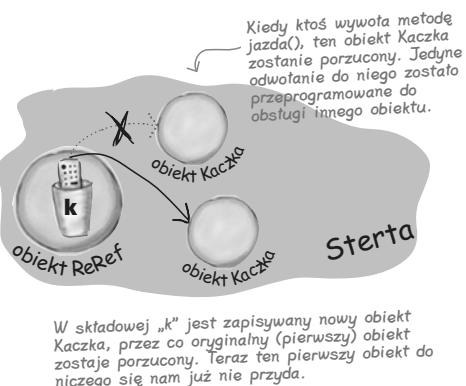
Abstrakcyjne kontra konkretne	230
Metody abstrakcyjne	231
Polimorfizm w działaniu	234
Czym jest „superultramiegaklasa” Object?	237
Stosowanie polimorficznych odwołań typu Object ma swoją cenę...	239
Kiedy Pies nie zachowuje się jak Pies	240
Object nie szczeka	241
Połącz się ze swoim wewnętrznym Object-em	242
A co, jeśli musimy zmienić kontrakt?	246
Na pomoc spieszają interfejsy!	252



9

Życie i śmierć obiektu

Obiekty się rodzą i umierają. To Ty jesteś za to odpowiedzialny. To Ty decydujesz, kiedy i jak skonstruować obiekt. Ty także decydujesz, kiedy go *porzucić*. **Odśmieczacz pamięci** odzyska pamięć zajmowaną przez niepotrzebne obiekty. Przyjrzymy się, w jaki sposób obiekty są tworzone, gdzie „żyją” i w jaki sposób efektywnie je przechowywać lub porzucać. Oznacza to, że będziemy dyskutować o stercie, stosie, zasięgach, konstruktorach, konstruktorach nadzędnych, odwołaniach pustych oraz przydatności odśmieczacza pamięci.



Stos i sterta. Gdzie są przechowywane informacje?	264
Metody są zapisywane na stosie	265
A co ze zmiennymi lokalnymi, które są obiektami?	266
Jeśli zmienne lokalne są przechowywane na stosie, to gdzie są przechowywane składowe?	267
Cud utworzenia obiektu	268
Tworzenie obiektu Kaczka	270
Inicjalizacja stanu nowego obiektu Kaczka	271
Nanopregląd. Cztery rzeczy o konstruktorach, które należy zapamiętać	277
Znaczenie konstruktorów klasy bazowej w życiu obiektu	279
Konstruktorzy klas bazowych pobierające argumenty	283
Wywoływanie jednego przeciążonego konstruktora z poziomu innego	284

10

Liczby mają znaczenie

Zabaw się w matematyka. Java API udostępnia metody do wyznaczania wartości bezwzględnej, zaokrąglania liczb, określania wartości minimalnej i maksymalnej i tak dalej. A co z formatowaniem? Moglibyśmy chcieć wyświetlać liczby ze znakiem dolara na początku i dwoma miejscami dziesiętnymi. Albo wyświetlić daną w formie daty. Albo daty w zapisie stosowanym w Anglii. A co z przekształcaniem łańcucha znaków na liczbę? Lub zamianą liczby na łańcuch znaków? Zaczniemy jednak od wyjaśnienia, co oznacza, że zmienne lub właściwości są *statyczne*.



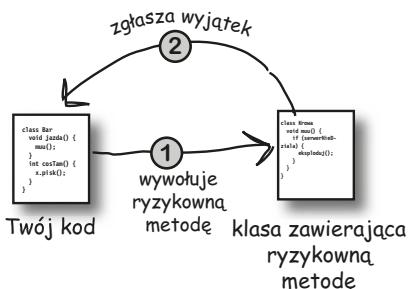
Metody klasy Math — najlepsze z możliwych odpowiedników metod globalnych	302
Różnice pomiędzy metodami zwyczajnymi a statycznymi	303
Co oznacza, że klasa ma statyczne metody	304
Składowa statyczna — ta sama wartość we wszystkich obiektach danej klasy	307
Inicjalizacja składowych statycznych	309
Role stałych pełnią statyczne zmienne finalne	310
Nie tylko zmienne statyczne mogą być finalne	311
Formatowanie liczb	322
Specyfikator formatu	326
Operacje na datach	330
Stosowanie obiektów Calendar	333
Wybrane możliwości API klasy Calendar	334

11

Rzykowne działania

Czasami zdarzają się nieprzewidziane sytuacje. Pliku nie ma tam, gdzie powinien być. Serwer został wyłączony. Niezależnie od tego, jak dobrym jesteś programistą, nie jesteś w stanie kontrolować *wszystkiego*. Kiedy tworzysz metodę, której działanie jest opatrzone ryzykiem niepowodzenia, musisz także stworzyć kod, który obsługuje potencjalnie niebezpieczną sytuację. Ale skąd wiadomo, że metoda może mieć ze sobą potencjalne niebezpieczeństwo?

Gdzie umieszczać kod *obsługujący wyjątkowe* sytuacje? W tym rozdziale stworzymy odtwarzacz MIDI wykorzystujący rzykowny interfejs programistyczny JavaSound, zatem lepiej dowiedzmy się, jak zabezpieczyć się przed potencjalnymi niebezpieczeństwstwami.



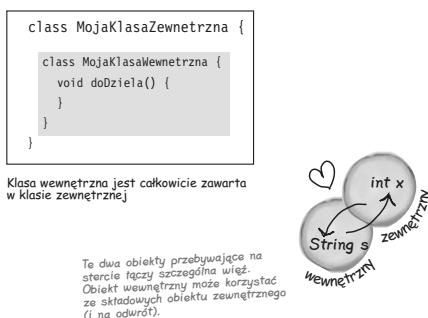
Stwórzmy program MuzMachina	344
JavaSound API	345
Wyjątek jest obiektem... klasy Exception	350
Finally — blok kodu, który musi zostać wykonany niezależnie od wszystkiego	355
Przechwytywanie wielu wyjątków	357
Reguły związane ze stosowaniem wyjątków	366
Generowanie dźwięków	368
Twój pierwszy odtwarzacz muzyki	370
Tworzenie obiektów MidiEvent (danych piosenki)	371
Komunikat MIDI — serce zdarzenia MidiEvent	372
Zmiana komunikatu	373

12

Historia bardzo graficzna

Musisz się z tym pogodzić — tworzenie interfejsów graficznych jest konieczne.

Nawet jeśli wierzysz, że już do końca życia będziesz pisać programy działające na serwerze, to jednak będziesz także musiał pisać programy narzędziowe i zapewne będziesz chciał, aby miały one jakiś interfejs graficzny. Zagadnieniom interfejsu graficznego poświęcimy dwa rozdziały, w których przedstawimy także inne cechy języka, takie jak obsługa zdarzeń oraz klasy wewnętrzne. Naciśniemy przycisk na ekranie, wskażemy coś myszą, wyświetlimy obraz zapisany w formacie JPEG, a nawet stworzymy małą animację.



Wszystko zaczyna się od okna	380
Twój pierwszy interfejs graficzny — przycisk w ramce	381
Przechwytywanie zdarzeń generowanych przez działania użytkownika	383
Odbiorcy, źródła i zdarzenia	387
Stwórz własny komponent umożliwiający rysowanie	390
Odwołanie do porządkowej klasy Graphics ukrywa obiekt Graphics2D	392
Układy GUI — wyświetlanie w ramce więcej niż jednego komponentu	396
Jak stworzyć obiekt klasy wewnętrznej?	404
Wykorzystanie klas wewnętrznych do tworzenia animacji	408
Odbieranie zdarzeń niezwiązanych z interfejsem użytkownika	413

13

Popracuj nad Swingiem

Swing jest łatwy. Chyba że naprawdę zwracasz uwagę na to, co jest gdzie wyświetlane. Kod wykorzystujący Swing *wydaje się prosty*, ale kiedy go skompilujesz, uruchomisz i popatrzyisz na wyniki, to często będziesz mógł sobie pomyśleć: „Hej, przecież to nie miało być wyświetcone w tym miejscu”. To, co zapewnia prostotę kodu, sprawia jednocześnie, że trudne jest kontrolowanie położenia elementów — tym „czymś” jest **menedżer układu**. Jednak przy odrobinie pracy można nagiąć menedżera układu do naszej woli. W tym rozdziale popracujemy nad naszym Swingiem i dowiemy się czegoś więcej o różnych elementach graficznych.



Komponenty na wschodzie i zachodzie mają preferowaną szerokość.

Także komponenty na południu i północy i południu mają preferowaną wysokość.

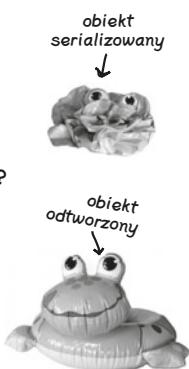
Komponenty biblioteki Swing	426
Komponenty można zagnieźdzać	426
Menedżery układu	427
W jaki sposób menedżery układu podejmują decyzje?	428
Różne menedżery układu mają różne zasady rozmieszczania	428
Wielka trójka menedżerów układu: BorderLayout, FlowLayout oraz BoxLayout	429
Zabawy z komponentami biblioteki Swing	439
Tworzenie aplikacji MuzMachina	445

14

Zapisywanie obiektów

Obiekty można pakować i odtwarzać. Obiekty mają swój stan i działanie. Działanie obiektu określa jego klasa, natomiast *stan* zależy od konkretnego *obiektu*. Jeśli program musi zapisać stan obiektu, możesz to zrobić w sposób trudny — analizując każdy obiekt i pracowicie zapisując wartości wszystkich właściwości. **Możesz także zapisać obiekt w sposób łatwy i obiektowy** — „zamrażając” obiekt (przeprowadzając jego serializację), a następnie odtwarzając (poprzez jego deserializację).

jakieś pytania?

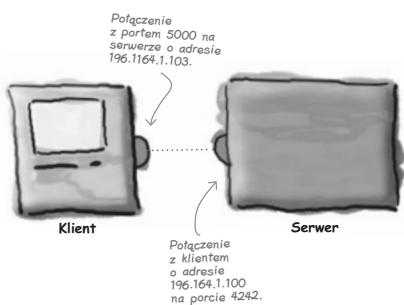


Odczytywanie taktów	454
Zapisywanie stanu	455
Zapisywanie serializowanego obiektu do pliku	456
Deserializacja — odtwarzanie obiektów	465
Zapisywanie łańcucha znaków w pliku tekstowym	471
Klasa java.io.File	476
Odczyt zawartości pliku tekstowego	478
Przetwarzanie łańcuchów znaków przy użyciu metody split()	482
Identyfikator wersji — wielki problem serializacji	484
Stosowanie serialVersionUID	485
Zapisywanie kompozycji	487

15

Nawiąż połączenie

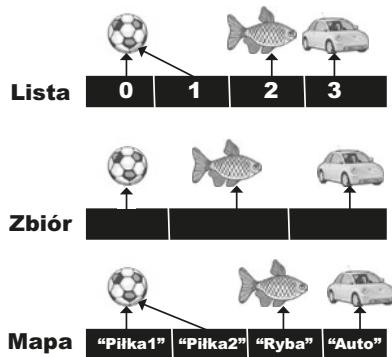
Nawiąż połączenie ze światem zewnętrznym. To takie łatwe. Wszelkie szczegóły związane z komunikacją sieciową na niskim poziomie są obsługiwane przez klasy należące do biblioteki `java.net`. Jedną z najlepszych cech Javy jest to, iż wysyłanie i odbieranie danych przez sieć to w zasadzie normalne operacje wejścia-wyjścia, z tą różnicą, że są w nich wykorzystywane nieco inne strumienie. W tym rozdziale stworzymy gniazda używane przez programy klienckie. Stworzymy także gniazda używane przez programy pełniące funkcje serwerów. Napiszemy zarówno program klienta, jak i serwera. Zanim skończysz czytać ten rozdział, będziesz już dysponować w pełni funkcjonalnym, wielowątkowym programem do prowadzenia internetowych pogawędek. Zaraz... czy właśnie padło słowo *wielowątkowy*?



Muzyczne pogawędki w czasie rzeczywistym	496
Nawiązywanie połączenia, wysyłanie i odbieranie danych	498
Nawiązanie połączenia sieciowego	499
Port TCP to tylko numer. 16-bitowa liczba identyfikująca konkretny program na serwerze	500
Aby odczytywać dane z gniazda, należy użyć strumienia <code>BufferedReader</code>	502
Aby zapisać dane w gnieździe, użąd strumienia <code>PrintWriter</code>	503
Program <code>CodziennePoradyKlient</code>	504
Kod programu <code>CodziennePoradyKlient</code>	505
Tworzenie prostego serwera	507
Kod aplikacji <code>CodziennePoradySerwer</code>	508
Tworzenie klienta pogawędek	510
Możliwość stosowania wielu wątków w Javie zapewnia jedna klasa — <code>Thread</code>	514
Jakie są konsekwencje posiadania więcej niż jednego stosu wywołań?	515
Aby stworzyć zadanie dla wątku, zaimplementuj interfejs <code>Runnable</code>	518
Mechanizm zarządzający wątkami	521
Usypanie wątku	525
Usypanie wątków w celu zapewnienia bardziej przewidywalnego działania programu	526
Tworzenie i uruchamianie dwóch wątków	527
Cóż... Tak, możemy. JEST pewien problem związany ze stosowaniem wątków	528
Problem Moniki i Roberta w formie kodu	530
Przykład Moniki i Roberta	531
Musimy wykonać metodę <code>pobierzGotowke()</code> jako operację atomową	534
Stosowanie blokady obiektu	535
Przerażający problem „utraconej modyfikacji”	536
Wykonajmy ten przykładowy kod...	537
Zadeklaruj metodę <code>inkrementuj()</code> jako metodę atomową. Synchronizuj ją!	538
Mroczna strona synchronizacji	540
Nowa i poprawiona wersja programu <code>ProstyKlientPogawedek</code>	542
Naprawdę prosty serwer pogawędek	544

16

Struktury danych



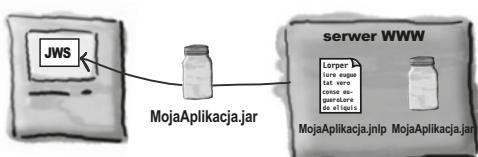
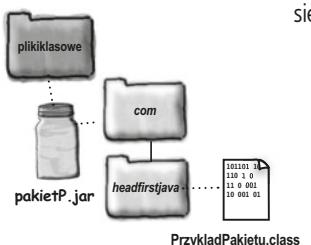
Sortowanie w Javie to pestka. Dysponujesz wszystkimi narzędziami do gromadzenia danych i manipulowania nimi, i to bez konieczności pisania własnych algorytmów. Biblioteka kolekcji Javy (*Java Collections Framework*) posiada struktury danych, które powinny sprostać praktycznie wszystkiemu, co będziesz chciał zrobić. Potrzebujesz listy, do której będziesz mógł łatwo dodawać elementy? Chcesz znaleźć coś na podstawie nazwy? A może chcesz stworzyć listę, która automatycznie będzie usuwać powtarzające się elementy? Albo posortować listę pracowników na podstawie liczby „przyjacielskich klepnięć w plecy”?

ArrayList nie jest jedyną dostępną kolekcją	557
Deklaracja metody sort()	563
Poznajemy typy ogólne	565
Używanie KLAS uogólnionych	566
Stosowanie METOD uogólnionych	568
Ponownie odwiedzimy metodę sort()	571
Stosowanie własnych komparatorów	576
Potrzebujemy zbioru, a nie listy	581
Biblioteka kolekcji (fragment)	582
Stosowanie argumentów polimorficznych i typów ogólnych	593
Znaki wieloznacznego śpieszą z pomocą	598

17

Rozpowszechnij swój kod

Już czas wypływać na „szerokie wody”. Napisałś kod swojej aplikacji. Przetestowałeś go. Udoskonaliłeś. Powiedziałś wszystkim znajomym, że świetnie by było, gdybyś już w życiu nie musiał napisać choćby jednej linijki kodu. Ale w końcu stworzyłeś dzieło sztuki. Przecież Twój aplikacja działa! W ostatnich dwóch rozdziałach książki zajmiemy się zagadnieniami związanymi z organizowaniem, pakowaniem i wdrażaniem kodu. Przyjrzymy się możliwościom wdrażania lokalnego, mieszanego i zdalnego, w tym wykonywalnym plikom JAR, technologii Java Web Start, RMI oraz serwletom. Nie stresuj się! Niektóre z najbardziej niesamowitych rozwiązań, jakimi może się poszczęścić Java, są prostsze, niż można by przypuszczać.

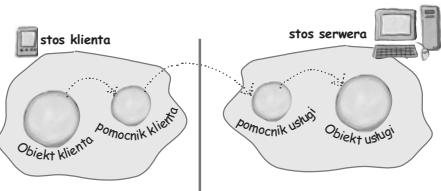


Wdrażanie aplikacji	606
Oddzielanie kodu źródłowego od plików klasowych	608
Umieszczanie programów w archiwach JAR	609
Uruchamianie (wykonywanie) archiwum JAR	610
Umieść klasy w pakietach	611
Zapobieganie konfliktom nazw pakietów	612
Kompilacja i uruchamianie programu, w którym wykorzystywane są pakiety	614
Flaga -d jest nawet lepsza, niż twierdziliśmy	615
Tworzenie wykonywalnego archiwum JAR zawierającego pakiety	616
Java Web Start	621
Plik .jnlp	623

18

Przetwarzanie rozproszone

Zdalne wykonywanie aplikacji nie zawsze jest złe. Oczywiście to prawda, że życie jest łatwiejsze, gdy wszystkie elementy aplikacji znajdują się w jednym miejscu i są zarządzane przez jedną wirtualną maszynę Javy. Jednak takie rozwiązanie nie zawsze jest możliwe. Co w sytuacji, gdy aplikacja realizuje bardzo złożone obliczenia? Co, jeśli potrzebuje informacji pochodzących z zabezpieczonej bazy danych? W tym rozdziale przedstawiona zostanie zadziwiająco prosta technologia wywoływania zdalnych metod — *RMI* (ang. *Remote Method Invocation*). Pobieżnie przyjrzymy się także innym rozwiązaniom — serwletom, komponentom *EJB* (ang. *Enterprise Java Bean*) oraz technologii Jini.



RMI udostępnia obiekty pomocnicze klienta i serwera	636
Bardzo prosty serwlet	649
Tak dla zabawy przeróbmy nasz program krasomówczy na serwlet	651
Enterprise JavaBeans — RMI na środkach dopingujących	653
I na samym końcu przedstawiamy... małego dżina Jini	654
Odkrywanie adaptacyjne w akcji	655

A

Dodatek A

Ostatnie doprawianie kodu. Kompletny kod aplikacji MuzMachina w ostatecznej wersji klient-serwer z możliwością prowadzenia muzycznych pogawędek. Twoja szansa, by zostać gwiazdą rocka.



Ostateczna wersja programu MuzMachina	672
Ostateczna wersja serwera aplikacji MuzMachina	679

B

Dodatek B

Dziesięć najważniejszych zagadnień, które niemal znalazły się w tej książce...
Jeszcze nie możemy Cię zostawić i wypuścić w świat. Mamy dla Ciebie kilka dodatkowych informacji, jednak to już jest koniec tej książki. I tym razem mówimy to zupełnie serio.

Lista dziesięciu zagadnień	682
----------------------------	-----

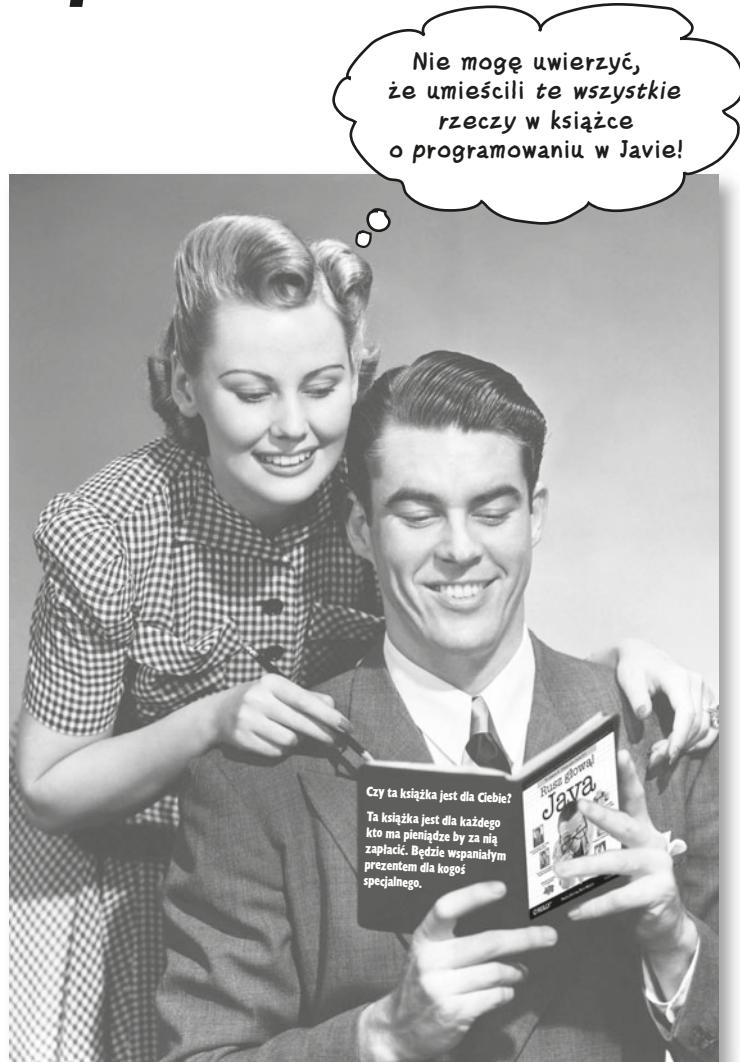
S

Skorowidz

699

Jak korzystać z tej książki?

Wprowadzenie



W tej części książki odpowiadamy na palcę pytanie:
„Dlaczego autorzy umieścili w książce te wszystkie
niezwykłe rzeczy?“.

Dla kogo jest przeznaczona ta książka?

Jeśli możesz odpowiedzieć twierdząco na *wszystkie* z poniższych pytań:

- ① Czy programowałeś już wcześniej?
- ② Czy chcesz nauczyć się programowania w Javie?
- ③ Czy wolisz stymulującą pogawędkę przy obiedzie niż taką i nudną techniczną publikację?

to jest to książka dla Ciebie.

To NIE jest podręcznik. Java. Rusz głowę! to książka zaprojektowana do nauki, a nie encyklopedia przedstawiająca fakty o Javie.

Kto prawdopodobnie nie powinien sięgać po tę książkę?

Jeśli możesz odpowiedzieć twierdząco na przynajmniej *jedno* z poniższych pytań:

- ① Czy Twoje doświadczenia programistyczne ograniczają się wyłącznie do języka HTML i to bez korzystania z jakichkolwiek języków skryptowych?
(Jeśli robiłeś cokolwiek wykorzystującego pętle lub logikę warunkową, to na pewno sobie poradzisz, jeśli jednak Twoje doświadczenia ograniczają się do stosowania znaczników języka HTML, to może to być nieco zbyt mało).
- ② Jesteś doświadczonym programistą C++ szukającym podręcznika Javy?
- ③ Boisz się spróbować czegoś nowego? Czy wolałbyś raczej leczenie kanałowe niż szal w szkocką kratkę. Czy uważasz, że książka o programowaniu, w której w dialecie o zarządzaniu pamięcią są rysunki kaczuszki, nie może być poważna?

to ta książka *nie jest* dla Ciebie.

[notatka z działu marketingu: kto usunął ten akapit o tym, że ta książka jest dla osób posiadających ważną kartę kredytową? I co z tą promocją wakacyjną „Podaruj przyjacielowi książkę o Javie”, o której rozmawialiśmy... — Franek]



Wiemy, co sobie myślisz

„Jakim cudem może *to* być poważna książka o programowaniu w Javie?”

„Po co te wszystkie obrazki?”

„Czy w taki sposób można się czegokolwiek *nauczyć*?”

„Czy czuję zapach pizzy?”



Twój mózg myśli, że właśnie
TO jest istotne

Wiemy także, co sobie myśli Twój mózg

Twój mózg pragnie nowości. Zawsze szuka, przegląda i *wyczekuje* czegoś niezwykłego. W taki sposób został stworzony i to pomaga mu przetrwać.

W dzisiejszych czasach jest mało prawdopodobne, abyś stał się przekąską dla tygrysa. Ale Twój mózg wciąż obserwuje. W końcu nigdy nic nie wiadomo.

Zatem co Twój mózg robi z tymi wszystkimi rutynowymi, zwyczajnymi, normalnymi informacjami, jakie do niego docierają? Otóż robi wszystko, co tylko *może*, aby nie przeszkadzały w jego najważniejszym *zadaniu* — zapamiętywaniu rzeczy, które mają prawdziwe *znaczenie*. Twój mózg nie traci czasu i energii na zapamiętywanie nudnych informacji; one nigdy nie przechodzą przez filtr „*to jest w oczywisty sposób całkowicie nieważne*”.



W jaki sposób Twój mózg *wie*, co jest istotne? Założmy, że jesteś na codziennej przechadzce i nagle przed Tobą staje tygrys, co się dzieje wewnątrz Twej głowy?

Neurony płoną. Emocje szaleją. *Adrenalina napływa falami*.

I właśnie dlatego Twój mózg wie, że...

To musi być ważne! Nie zapominaj o tym!!

Ale wyobraź sobie, że jesteś w domu albo w bibliotece. Jesteś w bezpiecznym miejscu — przytulnym i pozbawionym tygrysów. Uczysz się. Przygotowujesz się do egzaminu.

Albo rozgryzasz jakiś trudny problem techniczny, którego rozwiążanie, według szefa, powinno zająć Ci tydzień, a najdalej dziesięć dni.

Jest tylko jeden, drobny problem. Twój mózg stara się Ci pomóc.

Próbuje zapewnić, że te w oczywisty sposób *nieistotne* informacje nie zajmą cennych zasobów w Twojej głowie. Zasobów, które powinny zostać wykorzystane na zapamiętanie naprawdę *ważnych* rzeczy. Takich jak tygrysy. Takich jak zagrożenie, jakie niesie ze sobą pożar. Takie jak to, że już nigdy w życiu nie powinieneś jeździć na snowboardzie w krótkich spodenkach.

Co gorsze, nie ma żadnego sposobu, aby powiedzieć mózgowi:

„Hej mózgu mój, dziękuję ci bardzo, ale niezależnie od tego, jak nudna jest ta książka i jak nieznaczne są emocje, których aktualnie doznaję, to jednak *naprawdę* chciałbym zapamiętać wszystkie te informacje”.



Twój mózg uważa,
że tego nie warto
zapamiętywać.



Wyobrażamy sobie, że Czytelnik tej książki jest uczniem

Zatem, czego trzeba żeby się czegoś *nauczyć*? W pierwszej kolejności powinieneś to coś *poznać*, a następnie postarać się tego czegoś nie *zapomnieć*. I nie polega to jedynie na wtłoczeniu do głowy suchych faktów. Najnowsze badania prowadzone w dziedzinie przyswajania informacji, neurobiologii i psychologii nauczania pokazują, że *uczenie się wymaga czegoś więcej niż tylko czytania tekstu*. My wiemy, co potrafi pobudzić nasze mózgi do działania.

Oto niektóre z głównych założeń niniejszej książki:



Wyobraź to sobie wizualnie. Rysunki są znacznie łatwiejsze do zapamiętania niż same słowa i sprawiają, że uczenie staje się zdecydowanie bardziej efektywne (studia nad przypominaniem sobie i przekazywaniem informacji wykazują, że użycie rysunków poprawia efektywność zapamiętywania o 89%).

Poza tym rysunki sprawiają, że informacje stają się znacznie bardziej zrozumiałe. Wystarczy **umieścić słowa bezpośrednio na lub w okolicach rysunku**, do którego się odnoszą, a nie na następnej stronie, a prawdopodobieństwo, że osoby uczące się będą w stanie rozwiązać problem, którego te słowa dotyczą, wzrośnie niemal dwukrotnie.



Bycie metodą abstrakcyjną to naprawdę nic mitego. Wyobraź sobie, że jesteś zupełnie pusty w środku.



abstract void spaceruj();

Metoda nie ma treści! Kończy się średnikiem.

Zdobądź — przyciągnij na dłużej — uwagę i zainteresowanie Czytelnika. Każdy znalazł się kiedyś w sytuacji, gdy bardzo chciał się czegoś nauczyć, lecz zasypiał po przeczytaniu pierwszej strony. Mózg zwraca uwagę na rzeczy niezwykłe, interesujące, dziwne, przykuwające wzrok, nieoczekiwane. Jednak poznawanie nowego zagadnienia technicznego wcale nie musi być nudne. Jeśli będzie to zagadnienie interesujące, Twój mózg przyswoi je znacznie szybciej.

Czy ma sens stwierdzenie, że Wanna JEST Łazienką? A może Łazienka JEST Wanna? A może w tym przypadku lepsze jest użycie relacji MA?



Wyzwól emocje. Teraz już wiemy, że zdolności do zapamiętywania informacji są w znacznej mierze zależne od ich zawartości emocjonalnej. Zapamiętujemy to, na czym nam zależy. Zapamiętujemy w sytuacjach, w których coś odczuwamy. Oczywiście nie mamy tu na myśli wzruszających historii o chłopcu i jego psie. Chodzi nam o emocje takie jak zaskoczenie, ciekawość, radosne podekscytowanie, „a niech to...” i uczucie satysfakcji — „Jestem wielki!” — jakie odczuwamy po poprawnym rozwiązaniu zagadki, nauczeniu się czegoś, co powszechnie uchodzi za trudne lub zdaniu sobie sprawy, że znamy więcej szczegółów technicznych niż Zenek z działu inżynierii.



Metapoznanie — myślenie o myśleniu

Jeśli naprawdę chcesz się czegoś nauczyć i jeśli chcesz się tego nauczyć szybciej i dokładniej, to zwracaj uwagę na to, jak Ci na tym zależy. Myśl o tym, jak myślisz. Poznawaj sposób, w jaki się uczysz.

Większość z nas w okresie dorastania nie uczestniczyła w zajęciach z metapoznania albo teorii nauczania. *Oczekiwano* od nas, że będziemy się uczyć, jednak nie *uczono* nas, jak mamy to robić.

Jednak zakładamy, że jeśli trzymasz w ręku tę książkę, to chcesz nauczyć się Javy. I prawdopodobnie nie chcesz na to tracić zbyt wiele czasu.

Aby w jak największym stopniu wykorzystać tę książkę, *dowolną inną książkę lub jakikolwiek inny sposób uczenia się*, należy wziąć odpowiedzialność za swój mózg. Myśl o tym, czego się uczysz.

Sztuczka polega na tym, aby przekonać mózg, że poznawany materiał jest Naprawdę Ważny. Kluczowy dla Twojego dobrego samopoczucia. Tak ważny jak tygrys stojący naprzeciw Ciebie. W przeciwnym razie będziesz prowadzić nieustającą wojnę z własnym mózgiem, który ze wszystkich sił będzie się starać, aby nowe informacje nie zostały utrwalone.

A zatem, w jaki sposób zmusić mózg, aby potraktował Javę jak głodnego tygrysa?

Można to zrobić w sposób wolny i męczący lub szybki i bardziej efektywny. Wolny sposób polega na wielokrotnym powtarzaniu. Oczywiście wiesz, że jesteś w stanie nauczyć się i zapamiętać nawet najnudniejsze zagadnienie, mozołnie je „wkuwając”. Po odpowiedniej ilości powtórzeń Twój mózg stwierdzi: „*Wydaje się, że to nie jest dla niego szczególnie ważne, lecz w kółko to czyta i powtarza, więc przypuszczam, że jakąś wartość to jednak musi mieć*”.

Szybszy sposób polega na zrobieniu *czegokolwiek, co zwiększy aktywność mózgu*, a zwłaszcza jeśli czynność ta wyzwoli kilka różnych typów aktywności. Wszystkie zagadnienia, o jakich pisaliśmy na poprzedniej stronie, są kluczowymi elementami rozwiązania i udowodniono, że wszystkie z nich potrafią pomóc w zmuszeniu mózgu do tego, aby pracował na Twoją korzyść. Na przykład, badania wykazują, że umieszczenie słów *na* opisywanych rysunkach (a nie w innych miejscach tekstu na stronie, na przykład w nagłówku lub wewnątrz akapitu) sprawia, że mózg stara się zrozumieć relację pomiędzy słowami a rysunkiem, a to z kolei zwiększa aktywność neuronów. Większa aktywność neuronów, to większa szansa, że mózg *uzna* informacje za warte zainteresowania i, ewentualnie, zapamiętania.

Prezentowanie informacji w formie konwersacji pomaga, gdyż ludzie zdają się wykazywać większe zainteresowanie w sytuacjach, gdy uważają, że biorą udział w rozmowie, gdyż oczekuje się od nich, że będą śledzić jej przebieg i brać w niej czynny udział. Zadziwiające jest to, iż mózg zdaje się nie zważyć na to, że rozmowa jest prowadzona z książką! Z drugiej strony, jeśli sposób przedstawiania informacji jest formalny i suchy, mózg postrzega to tak samo jak w sytuacji, gdy uczestniczysz w wykładzie na sali pełnej sennych studentów. Nie ma potrzeby wykazywania jakiejkolwiek aktywności.

Jednak rysunki i przedstawianie informacji w formie rozmowy to jedynie początek.

*Zastanawiam się,
jak zmusić mózg do
zapamiętania tych
informacji...*



Jak korzystać z tej książki?

Oto co zrobiliśmy:

Używaliśmy **rysunków**, ponieważ Twój mózg zwraca większą uwagę na obrazy niż na tekst. Jeśli chodzi o mózg, to faktycznie jeden obraz jest wart 1024 słów. W sytuacjach, gdy pojawiał się zarówno tekst, jak i rysunek, umieszczaliśmy tekst na rysunku, gdyż mózg działa bardziej efektywnie, gdy tekst jest *wewnętrzny* czegoś, co opisuje niż kiedy jest umieszczony w innym miejscu i stanowi część większego fragmentu tekstu.



Stosowaliśmy **powtórzenia**, wielokrotnie podając tę samą informację na różne sposoby i przy wykorzystaniu różnych środków przekazu oraz odwołując się do różnych zmysłów. Wszystko to po to, aby zwiększyć szansę, że informacja zostanie zakodowana w większej ilości obszarów Twojego mózgu.

Używaliśmy pomysłów i rysunków w **nieoczekiwany** sposób, ponieważ Twój mózg oczekuje i pragnie nowości, poza tym staraliśmy się zawiązać w nich *chociaż trochę emocji*, gdyż mózg jest skonstruowany w taki sposób, iż zwraca uwagę na biochemię związaną z emocjami. Prawdopodobieństwo zapamiętania czegoś jest większe, jeśli „to coś” sprawia, że coś *poczujemy*, nawet jeśli to uczucie nie jest niczym więcej jak lekkim **rozbawieniem, zaskoczeniem lub zainteresowaniem**.



Używaliśmy bezpośrednich zwrotów i przekazywaliśmy treści w **formie konwersacji**, gdyż mózg zwraca większą uwagę, jeśli sądzi, że prowadzisz rozmowę, niż gdy jesteś jedynie biernym słuchaczem prezentacji. Mózg działa w ten sposób nawet wtedy, gdy czytasz zapis rozmowy.

Zamieściliśmy w książce ponad 50 **ćwiczeń**, ponieważ mózg uczy się i pamięta więcej, gdy coś *robi*, niż gdy o czymś *czyta*. Poza tym podane ćwiczenia stanowią wyzwania, choć nie są przesadnie trudne, gdyż właśnie takie preferuje większość osób.



Stosowaliśmy **wiele stylów nauczania**, gdyż Ty możesz preferować instrukcje opisujące krok po kroku sposób postępowania, ktoś inny analizowanie zagadnienia opisanego w ogólny sposób, a jeszcze inne osoby — przejrzenie przykładowego fragmentu kodu. Jednak niezależnie od ulubionego sposobu nauki *każdy* skorzysta na tym, że te same informacje będą przedstawiane kilkukrotnie na różne sposoby.

Podawaliśmy informacje przeznaczone dla **obu półkul Twojego mózgu**, gdyż im bardziej mózg będzie zaangażowany, tym większe jest prawdopodobieństwo nauczenia się i zapamiętania podawanych informacji i tym dłużej możesz koncentrować się na nauce. Ponieważ angażowanie tylko jednej półkuli mózgu często oznacza, że druga będzie mogła odpocząć, zatem będziesz mógł uczyć się bardziej produktywnie przez dłuższy okres czasu.



Dodatkowo zamieszczaliśmy **opowiadania** i ćwiczenia prezentujące *więcej niż jeden punkt widzenia*, ponieważ mózg uczy się łatwiej, gdy jest zmuszony do przetwarzania i podawania własnej opinii.



Stawialiśmy przed Tobą **wyzwania**, zarówno poprzez podawanie ćwiczeń, jak i stawiając **pytania**, na które nie zawsze można odpowiedzieć w prosty sposób, a to dlatego, że mózg uczy się i pamięta, gdy musi *popracować* nad czymś (to tak samo, jak nie można zdobyć dobrej kondycji, obserwując ćwiczenia w telewizji). Jednak dołożyliśmy wszelkich starań, aby zapewnić, że gdy pracujesz, to robisz *dokładnie* to, co trzeba. Aby *ani jeden dendryt nie musiał* przetwarzać trudnego przykładu, ani analizować tekstu zbyt lapidarnego lub napisanego niezrozumiałym żargonem.

Zastosowaliśmy metodę **80/20**. Zakładamy bowiem, że to nie jest książka dla osób, które mają zamiar piisać doktorat na temat Javy. Zatem nie zajmujemy się w niej *wszelkimi* możliwymi zagadnieniami, a jedynie tymi, których faktycznie możesz *używać*.





Oto co możesz zrobić, aby zmusić swój mózg do posłuszeństwa

A zatem zrobiliśmy, co było w naszej mocy. Reszta zależy od Ciebie. Możesz zacząć od poniższych porad. Posłuchaj swojego mózgu i określ, które sprawdzają się w Twoim przypadku, a które nie dają pozytywnych rezultatów.

*Wytnij te porady
i przyklej na lodówce.*



① Zwolnij. Im więcej rozumiesz, tym mniej musisz zapamiętać.

Nie ograniczaj się jedynie do *czytania*. Przerwij na chwilę lekturę i pomyśl. Kiedy znajdziesz w tekście pytanie, nie zaglądaj od razu na stronę odpowiedzi. Wyobraź sobie, że ktoś faktycznie zadaje Ci pytanie. Im bardziej zmusisz swój mózg do myślenia, tym większa będzie szansa, że się nauczysz i zapamiętasz dane kwestie.

② Wykonuj ćwiczenia. Rób notatki.

Umieszczałyśmy je w tekście, jednak jeśli zrobilibyśmy je za Ciebie, to niczym nie różniłoby się to od sytuacji, w której ktoś za Ciebie wykonywałby ćwiczenia fizyczne. I nie ograniczaj się jedynie do *czytania* ćwiczeń. **Używaj ołówka.** Można znaleźć wiele dowodów na to, że fizyczna aktywność podczas nauki może poprawić jej wyniki.

③ Czytaj fragmenty oznaczone jako „Nie istnieją głupie pytania”.

Chodzi tu o wszystkie fragmenty umieszczone z boku tekstu. Nie są to fragmenty opcjonalne — stanowią one część podstawowej zawartości książki! Zdarza się, że pytania są bardziej cenne niż odpowiedzi.

④ Rób sobie przerwy.

Wstań, przeciągnij się, pochodź trochę, zmień fryzurę, przejdź do innego pokoju. Dzięki temu Twój mózg będzie mógł coś poczuć, a nauka nie będzie zbyt związana z konkretnym miejscem.

⑤ Niech lektura tej książki będzie ostatnią rzeczą, jaką robisz przed późnkiem spać. A przynajmniej ostatnią rzeczą stanowiącą wyzwanie intelektualne.

Pewne elementy procesu uczenia się (a w szczególności przenoszenie informacji do pamięci długoterminowej) następują po odłożeniu książki. Twój mózg potrzebuje trochę czasu dla siebie i musi dodatkowo przetworzyć dostarczone informacje. Jeśli podczas tego koniecznego na wykonanie dodatkowego „przetwarzania” czasu zmusisz go do innej działalności, to część z przyswojonych informacji może zostać utracona.

⑥ Pij wodę. Dużo wody.

Twój mózg pracuje najlepiej, gdy dostarcza mu się dużo płynów. Odwodnienie (które może następować nawet zanim poczujesz pragnienie) obniża zdolność percepji.

⑦ Rozmawiaj o zdobywanych informacjach. Na głos.

Mówienie aktywuje odmienne fragmenty mózgu. Jeśli próbujesz coś zrozumieć lub zwiększyć szansę na zapamiętanie informacji na dłużej, powtarzaj je na głos. Jeszcze lepiej — staraj się je na głos komuś wytłumaczyć. W ten sposób nauczysz się szybciej, a oprócz tego będziesz mógł odkryć kwestie, o których nie wiedziałaś podczas czytania tekstu książki.

⑧ Postuchoj swojego mózgu.

Uważaj, kiedy Twój mózg staje się przeciążony. Jeśli spostrzegasz, że zaczynasz czytać побieżnie i zapominać to, o czym przeczytałeś przed chwilą, to najwyższy czas, żeby sobie zrobić przerwę. Po przekroczeniu pewnego punktu, nie będziesz się uczył szybciej „wciskając” do głowy więcej informacji, co gorsze, może to zaszkodzić całemu procesowi nauki.

⑨ Poczuj coś!

Twój mózg musi wiedzieć, że to, czego się uczysz, ma znaczenie. Z zaangażowaniem śledź zamieszczone w tekście opowiadania. Nadawaj własne tytuły zdjęciom. Zalewanie się łzami ze śmiechu po przeczytaniu głupiego dowcipu i tak jest lepsze od braku jakiekolwiek reakcji.

⑩ Własnoręcznie wpisuj i wykonuj kod.

Wpisuj i wykonuj przykładowe programy podawane w tekście. Potem będziesz mógł eksperymentować, modyfikując je i usprawniając (lub też psując, co czasami jest nawet lepszym sposobem zobaczenia, co się tak naprawdę dzieje). W przypadku dłuższych fragmentów kodu oraz Przykładów gotowych do użycia, możesz pobrać pliki z serwera FTP wydawnictwa Helion — <ftp://ftp.helion.pl/przyklady/javrg2.zip>

Czego potrzebujesz, aby skorzystać z tej książki?

Nie będziesz potrzebować żadnych narzędzi programistycznych, takich jak zintegrowane środowiska programistyczne (IDE). Zalecamy, abyś przed zakończeniem lektury tej książki (a w szczególności przed przeczytaniem rozdziału 16.) *nie* używał żadnych programów za wyjątkiem prostego edytora tekstu. Zintegrowane środowiska programistyczne mogą ukryć przed Tobą niektóre szczegóły, które mają bardzo duże znaczenie, a zatem znacznie lepiej jest zaczynać naukę Javy, stosując narzędzia obsługiwane z poziomu wiersza poleceń, a dopiero potem, kiedy dobrze zrozumiesz, co się naprawdę dzieje, skorzystać z narzędzi automatyzujących niektóre zadania.



KONFIGUROWANIE JAVY

- Jeśli jeszcze nie masz **Java 2 Standard Edition SDK** (ang. Software Development Kit) w wersji **1.5** lub wyższej, to będziesz jej potrzebować. Jeśli używasz systemów Linux, Windows lub Solaris, to będziesz mógł pobrać Javę za darmo z witryny <http://www.oracle.com/us/technologies/java/index.html>. Przejście ze strony głównej witryny na stronę pozwalającą na pobranie oprogramowania J2SE wymaga zazwyczaj nie więcej niż dwóch kliknięć. Należy pobrać najnowszą wersję, która nie jest *wersją testową* (tak zwaną wersją „beta”). J2SE SDK zawiera wszystko, co jest niezbędne do skompilowania i wykonania programu napisanego w Javie.
- Jeśli korzystasz z Mac OS X 10.4, to pakiet Java SDK już jest zainstalowany na Twoim komputerze. Stanowi on część systemu operacyjnego Mac OS X i nie będziesz potrzebować *niczego więcej*. Jeśli używasz wcześniejszej wersji systemu Mac OS X, to będziesz miał zainstalowaną wcześniejszą wersję JDK, na której z powodzeniem będzie działać 95% kodu przedstawionego w tej książce.
Uwaga: Ta książka bazuje na wersji 1.5 języka Java, jednak z zadziwiająco niejasnych przyczyn marketingowych Sun zmienił nazwę języka na Java 5, pozostawiając jednocześnie „1.5” jako numer wersji JDK. A zatem możemy się spotkać z nazwami Java 1.5, Java 5, Java 5.0 albo „Tiger” (taka bowiem była oryginalna nazwa kodowa tej wersji Javy) — *wszystkie one oznaczają to samo*. Nigdy nie było Javy 3.0 ani 4.0 — numeracja przeskoczyła z 1.4 na 5.0, wciąż jednak można znaleźć miejsca, gdzie pojawia się wersja 1.5 zamiast 5. Nas o to nie pytaj. (A... żeby było jeszcze zabawniej, zarówno Java 5, jak i Mac OS X 10.4 miały tę samą nazwę kodową „Tiger”; a ponieważ do uruchomienia Javy 5 na komputerach Macintosh jest wymagany system Mac OS X 10.4, zatem można usłyszeć rozmowy o uruchamianiu „Tigera na Tigerze”... A chodź o uruchamianie Javy 5 w systemie Mac OS X 10.4).
- SDK *nie* zawiera **dokumentacji interfejsu programistycznego (API)** Javy, a dokumentacja ta jest niezbędna. A zatem należy jeszcze raz odwiedzić witrynę java.sun.com i pobrać dokumentację J2SE API. Dokumenty wchodzące w skład tej dokumentacji można także znaleźć bezpośrednio na witrynie i przeglądać bez pobierania na lokalny komputer, jednak rozwiązanie takie jest nieco uciążliwe. Uwierz nam, warto pobrać tę dokumentację.
- Będziesz także potrzebować **edytora tekstu**. Praktycznie można skorzystać z dowolnego edytora (*vi*, *emacs*, *pico*), w tym także z programów wyposażonych w graficzny interfejs użytkownika, które można znaleźć w większości systemów operacyjnych. Wszystkie edytory, takie jak Notatnik, Wordpad,TextEdit i tak dalej, będą się nadawać do pisania programów w Javie, o ile upewnisz się, że do nazwy pliku z kodem źródłowym nie zostanie dodane rozszerzenie „.txt”.
- Po pobraniu i rozpakowaniu Java SDK (lub innym przygotowaniu, które może być różne w zależności od używanego systemu operacyjnego), należy dodać do zmiennej środowiskowej **PATH** informacje określające położenie katalogu */bin* znajdującego się w głównym katalogu SDK. Na przykład, jeśli J2SDK zostanie umieszczony na dysku w katalogu „*j2sdk1.5.0*”, to wewnątrz niego poszukaj katalogu „*bin*”, który zawiera wszystkie narzędzia (programy) niezbędne do korzystania z Javy. Musisz określić ścieżkę dostępu właśnie do tego katalogu, tak aby po wpisaniu w wierszu poleceń systemu komendy:

> **javac**

system wiedział, gdzie należy szukać kompilatora *javac*.

Notatka: Jeśli masz jakieś problemy z zainstalowaniem Javy, zajrzyj na jedno z wielu forów dyskusyjnych; polecamy na przykład forum Java-Beginning na portalu javaranch.com lub sekcję poświęconą Javie na polskojęzycznym forum 4programmers.net (<http://forum.4programmers.net/>).



Kilka ostatnich spraw, o których musisz wiedzieć

To książka dla osób zaczynających naukę Javy, a nie podręcznik akademicki. Celowo usunęliśmy wszystko, co mogłoby Ci przeszkadzać w *nauce*, niezależnie od tego, nad czym pracujesz w danym miejscu książki. Podczas pierwszej lektury, należy zaczynać od jej samego początku, gdyż kolejne rozdziały bazują na tym, co widziałeś i czego się dowiedziałeś wcześniej.

Używamy prostych diagramów wzorowanych na UML.

Gdybyśmy użyli UML-a w jego *właściwej* postaci, to w książce zobaczyłbyś kod, który *przypomina* Javę, lecz którego składnia jest po prostu błędna. A zatem wykorzystaliśmy uproszczoną wersję UML-a, która nie koliduje ze składnią Javy. Jeśli jeszcze nie znasz UML-a, to nie będziesz musiał się przejmować uczeniem się *obu* tych języków jednocześnie.

Nie zwracamy uwagi na organizację ani spakowanie kodu przykładów, aż do samego końca książki.

W tej książce możesz się skoncentrować na nauce języka bez konieczności stresowania się i zwracania uwagi na szczegóły związane z organizacją i administracją programami pisanyymi w Javie. W praktyce *będziesz* musiał poznać te informacje i korzystać z nich, dlatego też szczegółowo je opiszymy. Niemniej jednak prezentacja tych zagadnień została zamieszczona dopiero na samym końcu książki (w rozdziale 17.). Zrelaksuj się.

Ćwiczenia podawane na końcu każdego z rozdziałów są obowiązkowe, natomiast zagadki możesz rozwiązywać opcjonalnie. Na końcu każdego z rozdziałów zostały podane zarówno odpowiedzi na pytania, jak i rozwiązania zagadek.

O zagadkach musisz wiedzieć jedną rzeczą — *to są zagadki*. Tak jak zagadki logiczne, szarady, krzyżówki i tak dalej. Ćwiczenia zostały podane, aby pomóc Ci w przećwiczeniu poznanych informacji i dlatego powinieneś wykonać je wszystkie. Z kolei zagadki to zupełnie inną historią, a niektóre z nich, jako zagadki, stanowią całkiem poważne wyzwanie. Są one przeznaczone dla osób *pasjonujących* się zagadkami, a prawdopodobnie już wiesz, czy jesteś jedną z nich czy nie. Jeśli jednak nie jesteś tego pewny, to sugerujemy, abyś spróbował rozwiązać kilka z tych zagadek. Niezależnie od rezultatów, nie zniechęcaj się, jeśli *nie będziesz* w stanie rozwiązać którejś z nich, lub jeśli nie masz na tyle czasu, aby nad nimi popracować.

Do ćwiczeń oznaczonych jako „Zaostrz ołówek” nie ma odpowiedzi.

A przynajmniej nie zostały one opublikowane w niniejszej książce. W niektórych przypadkach nie można podać żadnej poprawnej odpowiedzi, z kolei w innych doświadczenia wyniesione z rozwiązywania zadań zamieszczonych w tych częściach książki pozwolą Ci określić, czy podane odpowiedzi są dobre czy nie.

Przykładowe kody są maksymalnie zwięzłe.

Przeglądanie 200 wierszy kodu w poszukiwaniu dwóch linijek, które należy zrozumieć, może być frustrujące. W większości przykładów zamieszczonych w tej książce dodatkowy kod, który nie jest bezpośrednio związany z omawianymi zagadnieniami, został w jak największym stopniu skrócony. Dzięki temu fragmenty, których musisz się nauczyć, są przejrzyste i proste. Nie należy zatem oczekwać, że podawane przykłady będą solidne ani nawet, że będą kompletne. To będzie Twoje zadanie, po zakończeniu lektury tej książki. Przykłady te zostały stworzone z myślą o *nauce* języka i ich możliwości funkcjonalne nie zawsze są kompletne.

Wykorzystaliśmy zmodyfikowany, uproszczony sposób zapisu przypominający UML

PIES
imie
szczeekaj()
jedz()
gonkota()

Powinieneś wykonać wszystkie zadania oznaczone jako „Zaostrz ołówek”.



Zadania oznaczone jako „Ćwiczenie” są obowiązkowe! Jeśli poważnie traktujesz naukę Javy, nie powinieneś ich pomijać.



Ćwiczenie

Zadania oznaczone symbolem puzzla są opcjonalne. Jeśli nie lubisz zakręconej logiki ani krzyżówek, to także i te zadania nie przypadną Ci do gustu.



Redaktorzy techniczni

„Podziękowania należą się wszystkim, lecz odpowiedzialność za błędy spada jedynie na autorów...”. Czy ktokolwiek naprawdę w to wierzy? Czy widzicie te dwie osoby przedstawione na tej stronie? Jeśli znajdziecie w tej książce jakieś problemy techniczne, to prawdopodobnie będzie to *ich* wina :)



Jess pracuje w firmie Hewlett-Packard w zespole Self-Healing Services Team (samonaprawiających się usług). Ma tytuł licencjata inżynierii komputerowej uzyskany na Uniwersytecie Villanova, certyfikaty SCJP 1.4 oraz SCWCD i dosłownie w ciągu kilku miesięcy otrzyma tytuł magistra inżynierii oprogramowania na Uniwersytecie Drexel (o rany!).

Kiedy nie pracuje ani nie studiuje, ani nie jeździ swoim mini, można zobaczyć, jak walczy ze swoim kotem o kłybek przedzy w ramach kończenia jej ostatniego projektu z robótek na drutach (czy jest jakiś chętny na czapkę?). Jessica pochodzi z Salt Lake City, ze stanu Utah (nie, nie jest mormonką... i tak miałeś zamiar o to zapytać), a aktualnie mieszka w Filadelfii ze swym mężem Mendrą i dwoma kotami: Chai i Sake.

Można ją znaleźć na portalu javaranch.com, gdzie zajmuje się moderowaniem forów technicznych.

Valentin Valentin Crettaz posiada tytuł magistra nauk informatycznych i komputerowych uzyskany w Szwajcarskim Federalnym Instytucie Technologii (EPFL) w Lozannie. Pracował jako inżynier oprogramowania w firmie SRI International (Menlo Park, Kalifornia) oraz jako główny inżynier w zespole Laboratorium Inżynierii Oprogramowania EPFL.

Valentin jest współzałożycielem i kierownikiem do spraw technologicznych w firmie Condris Technologies, specjalizującej się w opracowywaniu rozwiązań z zakresu architektury oprogramowania.

Jego zainteresowania badawcze i programistyczne obejmują technologie zorientowane aspektowo, wzorce projektowe i architekturnalne, usługi sieciowe oraz architekturę oprogramowania. Oprócz dbania o żonę, zajmowania się ogrodem, czytania i uprawiania sportów, Valentin zajmuje się moderowaniem forów poświęconych certyfikatom SCBCD i SCDJWS na witrynie javaranch.com. Valentin posiada certyfikaty SCJP, SCJD, SCBCD, SCWCD oraz SCDJWS. Miał także okazję, by stać się współautorem symulatora egzaminu SCBCD firmy Whizlabs.

(Wciąż jesteśmy w szoku, że zobaczyliśmy go w krawacie).

pochwalić

Inne osoby, które można obwinić

W wydawnictwie O'Reilly:

Składamy ogromne podziękowania **Mike'owi Lukiedesowi** z wydawnictwa O'Reilly za skorzystanie z okazji i pomoc w wypracowaniu pomysłu na książkę i całą serię *Rusz głową!* Kiedy w druku ukazuje się niniejsze, drugie wydanie książki Java. *Rusz głową!*, cała seria niezwykle się rozrasta, a Mike był z nami cały ten czas. Dziękujemy **Timowi O'Reilly** za chęć zaangażowania się w coś całkowicie nowego i innego. Dziękujemy także mądrzej **Kyle Hurt** za określenie, jak seria *Rusz głową!* może zostać odebrana przez czytelników, oraz za rozpoczęcie prac nad nią. W końcu chcieliśmy także podziękować **Ediemu Freedmanowi** za zaprojektowanie okładki „wyraźnie podkreślającej znaczenie głowy”.

Nasza grupa nieustraszonych beta-testerów i recenzentów:

Największe honory i podziękowania chcieliśmy przekazać **Johannesowi de Jong** — dyrektorowi zespołu recenzentów technicznych portalu *javaranch.com*. Pomagałeś nam już po raz piąty przy książce z serii *Rusz głową!* i jesteśmy niezmiernie uradowani, że wciąż jeszcze z nami rozmawiasz. **Jeff Cumps** pomagał nam już przy trzeciej książce i niestrudzenie wynajduje fragmenty tekstu i zagadnienia, które powinniśmy poprawić lub wyjaśnić w bardziej zrozumiałym sposobie.

Coreyu McGone — jesteś super. I uważamy, że udzielasz najbardziej przejrzystych i zrozumiałych wyjaśnień na całym portalu *javaranch*. Pewnie zauważysz, że ukradliśmy kilka z nich. **Jason Menard** zaoszczędził nam wstydły z kilkoma technicznymi szczegółami, a **Thomas Paul**, jak zawsze, podzielił się swymi eksperckimi uwagami i znalazł subtelne problemy, które cała reszta z nas przegapiła. **Jane Griscti** także ma swój udział w powstawaniu tej książki (i wie co nieco o *pisaniu*) i było wspaniale, że mogła pomagać nam nad jej nowym wydaniem wraz z długoletnim pracownikiem portalu *javaranch.com* — **Barrym Guantem**.

Marilyn de Querioz bardzo nam pomogła w pracach nad *oboma* wydaniami tej książki. W pracach nad jej pierwszym wydaniem nieocenionej pomocy udzielili nam także **Chris Jones**, **John Nyquist**, **James Cubeta**, **Terri Cubeta** i **Ira Becker**.

Specjalne podziękowania chcieliśmy także przekazać kilku osobom, które z serią książek *Rusz głową!* są związane od samego początku; są to: **Angelo Celeste**, **Mikalai Zajkin** oraz **Thomas Duff** (*twduff.com*). Dziękujemy także naszemu wspaniałemu agentowi Davidowi Rogelbergowi z firmy Studio B (ale, ale... co z prawami do ekranizacji książki?).

Niektórzy z naszych eksperckich recenzentów...

*Jeff Cumps**Corey McGlone**Johannes de Jong**Jason Menard**Thomas Paul**Marilyn de Queiroz**Chris Jones**Rodney J. Woodruff**James Cubeta Terri Cubeta**Ira Becker**John Nyquist*

Kolejne podziękowania

W chwili, gdy pomyślałeś, że nie będzie już żadnych kolejnych podziękowań¹

Kolejni eksperci, którzy pomogli nam w pracach nad pierwszym wydaniem tej książki (wymienieni w pseudolosowej kolejności):

Emiko Hori, Michael Taupitz, Mike Gallihugh, Manish Hatwalne, James Chegwidden, Shweta Mathur, Mohamed Mazahim, John Paverd, Joseph Bih, Skulrat Patanavanich, Sunil Palicha, Sudhhasatwa Ghosh, Ramki Srinivasan, Alfred Raouf, Angelo Celeste, Mikalai Zaikin, John Zoetebier, Jim Pleger, Barry Gaunt, oraz Mark Dielen.

Zespół osób zajmujących się zagadkami z pierwszego wydania tej książki:

Dirk Schreckmann, Mary „JavaCross Champion” Leners, Rodney J. Woodruff, Gavin Bong, oraz Jason Menard. Portal javaranch.com naprawdę ma szczęście, że im pomagacie.

Inni współkonspiratorzy, którym chcieliśmy podziękować:

Paul Wheaton, Mistrz Podróży pomagający tysiącom osób pragnących nauczyć się Javy, odwiedzających portal javaranch.com. **Solveig Haugland** — mistrzyni J2EE i współautorka książki *Dating Design Patterns*. Autorom **Dori Smith** oraz **Tomowi Negrino** (backupbrain.com) za pomoc w nawigowaniu po świecie książek technicznych.

Współwinnym naszego „przestępstwa” — **Ericowi i Berh Freeman** (autorom książki *Wzorce projektowe. Rusz głową!*) — za drinki energetyczne Bawls™, dzięki którym ukończyliśmy tę książkę na czas.

Oraz **Sherry Dorris**, za rzeczy, które naprawdę mają znaczenie.

Dzielne osoby, które jako pierwsze zaczęły korzystać z serii Rusz głową!:

Joe Litton, Ross P. Goldberg, Dominic Da Silva, honestpuck, Danny Bromberg, Stephen Lepp, Elton Hughes, Eric Christensen, Vulinh Nguyen, Mark Rau, Abdulhaf, Nathan Oliphant, Michael Bradly, Alex Darrow, Michael Fischer, Sarah Nottingham, Tim Allen, Bob Thomas oraz Mike Bibby (on był pierwszy).

¹ Duża liczba podziękowań wynika z faktu, iż testujemy teorię, w myśl której każda osoba wymieniona w podziękowaniach kupi przynajmniej jeden egzemplarz książki, a może nawet więcej... w końcu co z rodziną i wszystkimi znajomymi? Jeśli chciałbyś się znaleźć w podziękowaniach w następnej naszej książce i masz dużą rodzinę — napisz.

1. Szybki skok na głęboką wodę

Przełamując zalew początkowych trudności

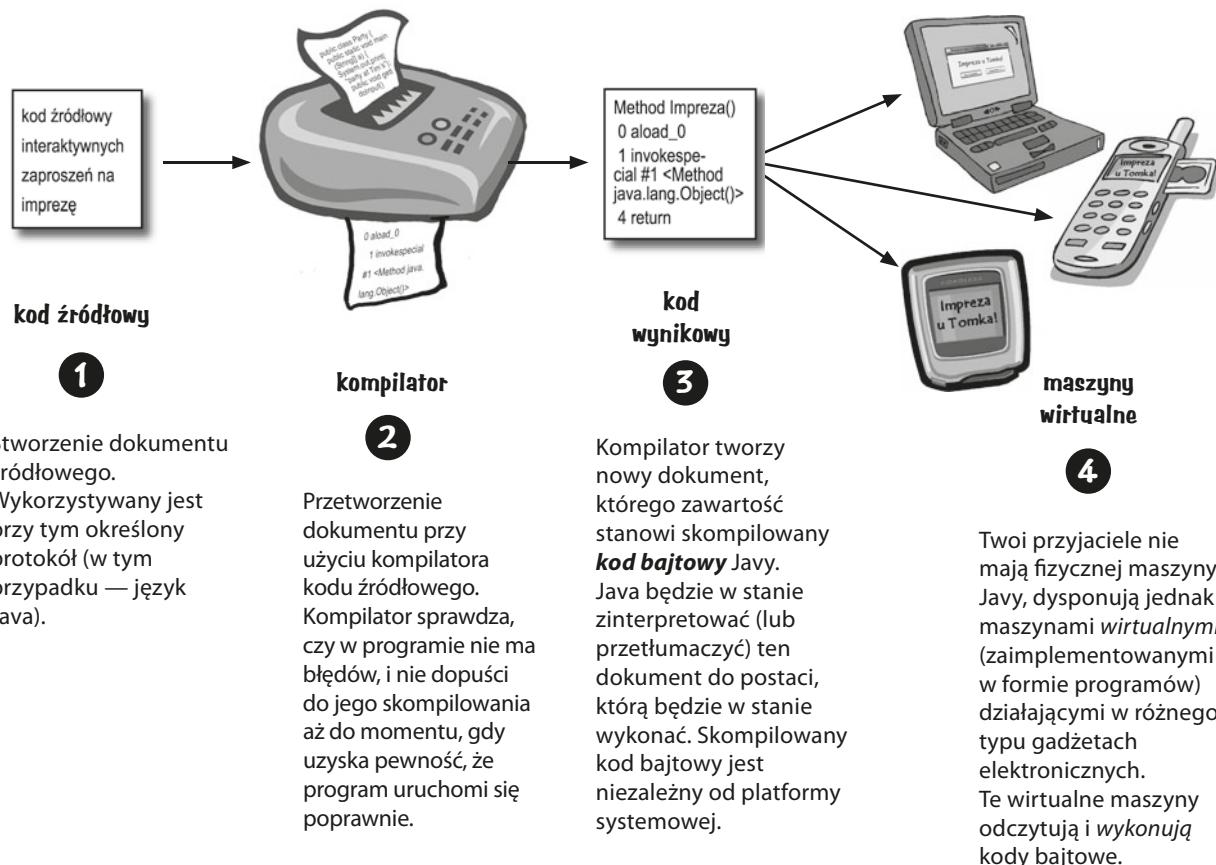


Java zabiera nas w nowe miejsca. Od momentu pojawienia się pierwszej, skromnej wersji o numerze 1.02 Java pociągała programistów ze względu na przyjazną składnię, cechy obiektowe, zarządzanie pamięcią, a przede wszystkim obietnicę przenośności. Pokusa, jaką niesie ze sobą hasło „**napisz raz — uruchamij wszędzie**”, jest po prostu zbyt duża. Nagle pojawiło się wielu oddanych użytkowników Javy, choć programiści walczyli z błędami, ograniczeniami a także faktem, że język był strasznie wolny. Ale tak było wieki temu. Jeśli właśnie zaczynasz naukę i korzystanie z Javy, **masz szczęście**. Niektórzy z nas musieli brnąć 10 kilometrów w śniegu, w obie strony pod górę (i to boso), aby uruchomić nawet najprostszy applet. Ale *Ty, Ty masz do dyspozycji udoskonaloną, szybszą i o niebo potężniejszą*, nowoczesną wersję języka.



Jak działa Java?

Naszym celem jest napisanie jednej aplikacji (w tym przykładzie będą to interaktywne zaproszenia na imprezę) i zagwarantowanie, aby mogła ona działać na dowolnych urządzeniach, które posiadają Twoi przyjaciele.



Co będziemy robić w Javie?

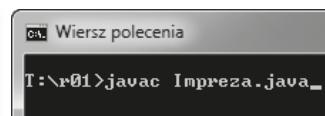
Napiszemy plik zawierający kod źródłowy, skompilujemy go, wykorzystując w tym celu kompilator javac, a następnie wykonamy skompilowany kod bajtowy przy użyciu wirtualnej maszyny Javy.

```
import java.awt.*;
import java.awt.event.*;
class Impreza {
    public static void tworzZaproszenie(){
        Frame f = new Frame();
        Label l = new Label("Impreza u Tomka");
        Button b = new Button("Się rozumie!");
        Button c = new Button("Zapomnij :)");
        Panel p = new Panel();
        p.add(l);
        //... dalsza część kodu
    }
}
```

kod źródłowy

1

Wpisz kod źródłowy.

Zapisz go jako
Impreza.java.

kompilator

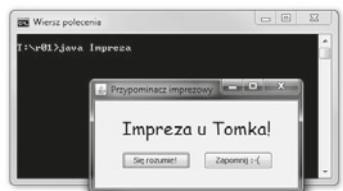
2

Skompiluj plik **Impreza.java**, uruchamiając program javac (czyli kompilator). Jeśli nie pojawią się żadne błędy, wygenerowany zostanie drugi plik o nazwie **Impreza.class**. Wygenerowany przez kompilator plik **Impreza.class** zawiera kod bajtowy.

```
Method Impreza()
0 aload_0
1 invokespecial #1 <Method
java.lang.Object()>
4 return
Method void tworzZaproszenie()
0 new #2 <Class java.awt.Frame>
3 dup
4 invokespecial #3 <Method
java.awt.Frame()>
```

kod wynikowy

3

Skompilowany kod:
Impreza.class.

maszyny wirtualne

4

Wykonaj program, uruchamiając w tym celu wirtualną maszynę Javy (ang. Java Virtual Machine, w skrócie JVM) i przekazując do niej plik **Impreza.class**. Wirtualna maszyna Javy przekształca kod bajtowy na polecenia, które jest w stanie zrozumieć system komputerowy, w jakim działa, a następnie wykonuje program.

(Uwaga: To nie ma być samouczek... już za chwilę będziesz piszą prawdziwy kod, jednak na razie chcemy, byś zrozumiał, jak te wszystkie elementy układanki pasują do siebie).

Bardzo krótka historia Javy

Ilość klas w standardowej bibliotece Javy

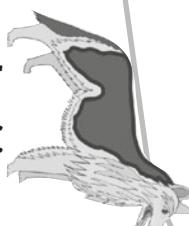
3500
3000
2500
2000
1500
1000
500
0



Java 1.02

250 klas

Wolna.
Fajna nazwa i logo.
Jej używanie dawało
dużo radości.
Masa błędów.
Zdecydowanie
najciekawsze są
apety.



Java 1.1

500 klas

Nieco szybsza.
Więcej możliwości, nieco
bardziej **przyjazna** dla
programistów. Staje się
bardzo **popularna**. Lepszы
kod do obsługi graficzne-
go interfejsu użytkownika
(GUI).



Java 2 (wersje 1.2 – 1.4)

2300 klas

Dużo szybsza.

Mожет (czasami) działać
z szybkością dorównującą
szybkości działania
programów komplikowanych
do postaci kodu

natywnego. Poważny
język programowania
o **ogromnych możliwościach**.

Dostępna w trzech wersjach:
Micro Edition (J2ME),
Standard Edition (J2SE) oraz
Enterprise Edition (J2EE). Stała

się **preferowanym językiem**
dla aplikacji biznesowych
(zwłaszcza internetowych)
oraz aplikacji dla urządzeń
przenośnych.



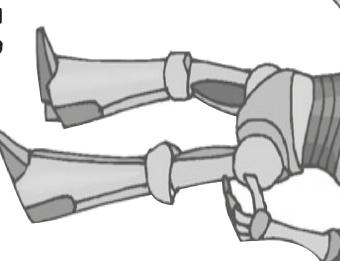
Java 5.0 (wersje 1.5 i kolejne)

3500 klas

**Ma jeszcze większe
możliwości i w większym
stopniu ułatwia pisanie
aplikacji.**

W Javie 5.0 (określanej także
jako „Tiger” — Tygrys) oprócz
dodania ponad tysiąca
nowych klas wprowadzono
także zmiany w samym

języku, które (przynajmniej
teoretycznie) miały ułatwiać
życie programistom
i rozszerzyć go o nowe
możliwości znane z innych,
popularnych języków
programowania.





Zaostrz ołówek: Rozwiążanie

Zobacz, jak łatwo jest napisać program w Javie.

```
int wielkosc = 27;  
  
String imie = "Azorek";  
  
Pies mojPies = new Pies(imie, wielkosc);  
  
x = wielkosc - 5;  
  
if (x < 15) mojPies.szczekaj(8);  
  
  
while (x > 3) {  
  
    mojPies.zabawa();  
  
}  
  
  
int[] listaNum = {2, 4, 6, 8};  
  
System.out.print("Witamy");  
  
System.out.print("Pies: " + imie);  
  
String liczba = "8";  
  
int z = Integer.parseInt(liczba);  
  
  
try {  
  
    czytajPlik("mojPlik.txt");  
  
}  
  
catch (FileNotFoundException ex) {  
  
    System.out.print("Nie znaleziono pliku.");  
  
}
```

Nie przejmuj się tym, czy cokolwiek z tych instrukcji rozumiesz, czy nie!

Wszystkie zostaną szczegółowo wyjaśnione w dalszej części książki (a większość z nich na pierwszych 40 stronach). Jeśli Java przypomina język programowania, którego używałeś wcześniej, to część z tych instrukcji zrozumiesz bez problemów. A jeśli nie zrozumiesz, to i tak się nie przejmuj. Wszystkim się zajmiemy...

Deklaruje zmienną całkowitą o nazwie „wielkosc” i przypisuje jej wartość 27.

Deklaruje zmienną łańcuchową o nazwie „imie” i przypisuje jej wartość „Azorek”.

Deklaruje zmienną typu Pies o nazwie „mojPies” i tworzy nowy obiekt klasy Pies, używając przy tym zmiennych „imie” i „wielkosc”.

Odejmuje liczbę 5 od 27 (wartości zmiennej „wielkosc”) i wynik zapisuje w zmiennej „x”.

Jeśli wartość zmiennej x (czyli 22) jest mniejsza od 15, to każde psu zaszczekać osiem razy.

Wykonuje pętlę tak długo, jak długo wartość zmiennej x jest większa od 3...

każe psu bawić się (cokolwiek to dla niego znaczy...)

to wygląda na koniec pętli — cały kod umieszczony pomiędzy nawiasami {} jest wykonywany w pętli.

Deklaruje zmienną „listaNum”, która będzie zawierać liczby całkowite, i zapisuje w niej liczby: 2, 4, 6 i 8.

Wyświetla słowo „Witamy”... najprawdopodobniej w wierszu poleceń.

Wyświetla tekst „Pies: Azorek” (zmienna „imie” ma wartość „Azorek”) w wierszu poleceń.

Deklaruje zmienną łańcuchową o nazwie „liczba” i zapisuje w niej znak „8”.

Konwertuje łańcuch zawierający znak „8” na liczbę o wartości 8.

Próbuje coś zrobić... jednak wcale nie ma gwarancji, że wykonanie operacji się powiedzie...

odczytuje plik tekstowy o nazwie „mojPlik.txt” (a przynajmniej próbuje to zrobić...)

ta próba kiedyś musi się skończyć, choć można by spróbować zrobić także inne rzeczy...

to chyba tu dowiadujemy się, czy nasza próba zakończyła się powodzeniem...

jeśli nie udało się wykonać zamierzanej operacji, w wierszu poleceń wyświetlany jest komunikat „Nie znaleziono pliku”.

wygląda na to, że wszystko, co umieszczono pomiędzy nawiasami {}, ma być wykonane w razie niepowodzenia operacji z bloku „try”...

Struktura kodu w Javie



Umieść definicję klasy w pliku źródłowym.

W klasie umieść definicje metod.

W metodach umieść instrukcje.

Co się umieszcza w pliku źródłowym?

Plik źródłowy (ten z rozszerzeniem `.java`) zawiera definicję **klasy**. Klasa reprezentuje pewien *element programu*, choć bardzo małe aplikacje mogą się składać tylko z jednej klasy. Zawartość klasy musi być umieszczona wewnątrz pary nawiasów klamrowych.

```
public class Pies {  
}  
klasa
```

Co się umieszcza w klasie?

Wewnątrz klasy umieszcza się jedną lub kilka **metod**. W klasie Pies metoda `szczekaj` mogłaby zawierać instrukcje określające, w jaki sposób pies ma szczekać. Wszystkie metody muszą być deklarowane *wewnątrz* klasy (innymi słowy, wewnątrz nawiasów klamrowych wyznaczających klasy).

```
public class Pies {  
    void szczekaj() {  
    }  
}  
metoda
```

Co się umieszcza w metodzie?

Wewnątrz nawiasów klamrowych metody należy umieszczać instrukcje określające sposób działania metody. *Kod* metody to po prostu zbiór instrukcji i, jak na razie, możesz wyobrażać sobie, że metoda to coś podobnego do funkcji lub procedury.

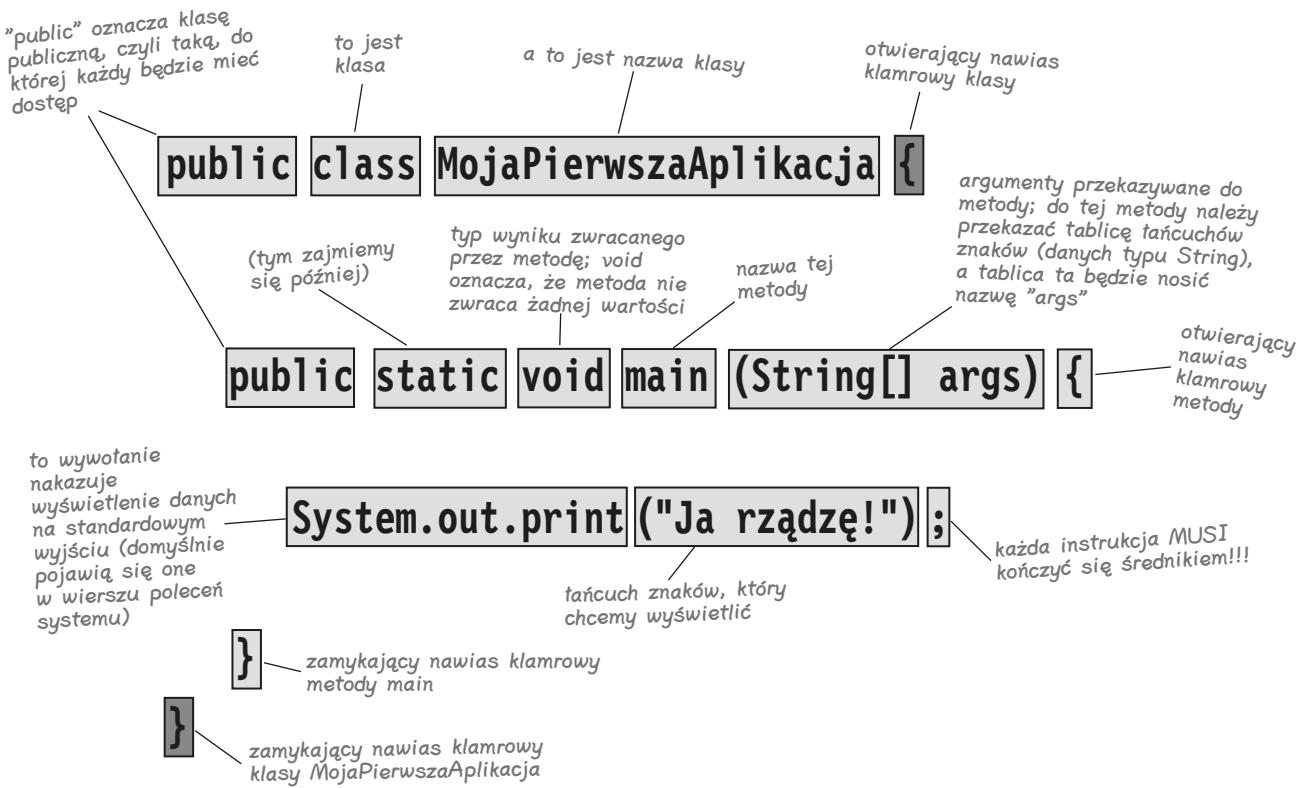
```
public class Pies {  
    void szczekaj() {  
        instrukcja1;  
        instrukcja2;  
    }  
}  
instrukcje
```

Anatomia klasy

Gdy JVM zaczyna działać, szuka pliku podanego w wierszu poleceń. Następnie poszukuje specjalnej metody, która wygląda dokładnie jak ta pokazana poniżej:

```
public static void main (String[] args) {  
    // tu umieszcza się kod metody  
}
```

Następnie JVM wykonuje wszystkie instrukcje zapisane pomiędzy nawiasami klamrowymi {} metody main(). Każda aplikacja Javy musi zawierać przynajmniej jedną **klasę** i przynajmniej jedną metodę **main** (zwróć uwagę, że nie chodzi o przynajmniej jedną metodę main na klasę, lecz na całą aplikację).



Nie zwracaj sobie teraz głowy zapamiętywaniem tego wszystkiego...

Ten rozdział ma Cię jedynie przygotować do rozpoczęcia pisania w Javie.

Tworzenie klasy z metodą main

W Javie wszystko jest zapisywane w **klasach**. W pierwszej kolejności należy stworzyć plik źródłowy (z rozszerzeniem *.java*), a następnie skompilować go do postaci nowego pliku klasowego (z rozszerzeniem *.class*). Uruchamiając program, tak naprawdę uruchamiamy *klasę*.

Uruchomienie programu oznacza wydanie wirtualnej maszynie Javy (JVM) polecenia: „Załaduj klasę **MojaPierwszaAplikacja**, a następnie zaczniż wykonywać metodę **main()**. Program ma działać aż do momentu wykonania całego kodu metody **main()**”.

W rozdziale 2. dokładniej zajmiemy się wszystkimi zagadnieniami związanymi z *klasami*, jednak jak na razie wystarczy, abyś wiedział, *jak napisać kod Javy, który będzie można poprawnie wykonać*. A to zagadnienie rozpoczyna się od utworzenia metody **main()**.

Działanie programu zaczyna się właśnie w metodzie **main()**.

Niezależnie od tego, jak duży jest program (innymi słowy, z ilu klas się składa), żeby zacząć całą zabawę, konieczna jest metoda **main()**.



Co można umieszczać w głównej metodzie programu?

Prawdziwa zabawa zaczyna się podczas tworzenia zawartości głównej metody programu — `main()` (jak również *wszelkich innych* metod). Można w niej umieścić wszelkie konstrukcje stosowane w większości innych języków programowania i które mają *sprawić, aby komputer coś zrobił*.

Kod może nakazać, aby wirtualna maszyna Javy:



1 Wykonała coś

Instrukcje: deklaracje, przypisania, wywołania metod i tak dalej

```
int x = 3;
String imie = "Azorek";
x = x + 17;
System.out.print("x = " + x);
double d = Math.random();
// a to jest komentarz
```

Składnia na wesoło

* Każda instrukcja musi się kończyć średnikiem.

`x = x + 1;`

* Komentarze jednowierszowe zaczynają się do dwóch znaków ukośnika.

`x = 22;`

// ten wiersz mi przeszkadza

* W większości przypadków odstępy nie mają znaczenia.

`x = 3 ;`

* Zmienne deklaruje się, podając **nazwę** oraz **typ** (wszelkie typy danych dostępne w Javie zostaną przedstawione w rozdziale 3.).

`int waga;`

// typ: int, nazwa: waga

* Definicje klas i metod należy umieszczać wewnętrz nawiasów klamrowych.

```
public void idz() {
    // tu znajdzie się zadziwiający kod
}
```

2 Wykonała coś wielokrotnie

Pętle: `for` oraz `while`

```
while (x > 12) {
    x = x - 1;
}

for (int x = 0; x < 10; x = x + 1) {
    System.out.print("aktualnie x = " + x);
}
```

3 Wykonała coś pod pewnym warunkiem

Rozgałęzienia: testy `if` oraz `if-else`

```
if (x == 10) {
    System.out.print("x musi mieć wartość 10");
} else {
    System.out.print("x jest różne od 10");
}
if ((x < 3) && (imie.equals("Azorek"))) {
    System.out.print("Nie rusz! Do nogi!");
}
System.out.print("Ten wiersz jest wykonywany niezależnie od wszystkiego!");
```



```
while (wiecejPilek == true) {
    zaglujDalej();
}
```

Pętle i pętle i...

Java udostępnia trzy podstawowe typy pętli: `while`, `do - while` oraz `for`. Pętle zostaną opisane dokładniej w dalszej części książki, ale jeszcze nie teraz, dlatego też na razie przedstawimy pętlę `while`.

Jej składnia (nie wspominając w ogóle o logice działania) jest tak prosta, że na pewno już usnęłaś z nudów. Wszystko, co jest umieszczone wewnętrz *bloku* pętli, jest wykonywane dopóty, dopóki pewien warunek logiczny jest spełniony. Blok pętli jest wyznaczany przez parę nawiasów klamrowych, a zatem wewnętrz nich powinien znaleźć się cały kod, który ma być powtarzany.

Kluczowe znaczenie dla działania pętli ma *test warunku*. W Javie testem takim jest wyrażenie, które zwraca wartość *logiczną* — innymi słowy coś, co może przyjąć wartość **true** (prawda) lub **false** (fałsz).

Jeśli napiszemy coś w stylu: „Dopóki (ang. *while*) w Kubeczkusią Lody jest prawdą, jedz dalej”, uzyskujemy typowy test logiczny. Istnieją tylko dwie możliwości: w kubeczku są lody lub ich *nie ma*. Ale jeśli napisalibyśmy: „Dopóki Janek kontynuuje konsumpcję”, nie uzyskalibyśmy prawdziwego testu. W takim przypadku należałoby zmienić warunek na coś w stylu: „Dopóki Janek jest głodny...” lub „dopóki Janek *nie jest* najedzony...”.

Proste testy logiczne

Prosty test logiczny można przeprowadzić, sprawdzając wartość zmiennej przy wykorzystaniu operatora porównania, takiego jak:

- < (mniejszy niż),
- > (większy niż),
- == (równy; tak, tak, to są dwa znaki równości).

Należy zwrócić uwagę na różnice pomiędzy operatorem przypisania (którym jest pojedynczy znak równości) oraz operatorem równości (zapisywany jako dwa znaki równości). Wielu programistów niechcąc wpisać = tam, gdzie w rzeczywistości chcieli umieścić ==. (Na pewno nie dotyczy to Ciebie).

```
int x = 4; // przypisujemy zmiennej x
            // wartość 4
while (x > 3) {
    // kod pętli zostanie wykonany, gdyż
    // x jest większe od 3
    x = x - 1; // w przeciwnym razie pętla
                // będzie wykonywana
                // w nieskończoność
}
int z = 27;
while (z == 17) {
    // kod pętli nie zostanie wykonany, gdyż
    // z nie jest równe 17
}
```

Nie istniejąca grupa pytań

P: Dlaczego wszystko musi być umieszczane wewnątrz klas?

O: Java jest językiem zorientowanym obiektowo. Nie przypomina w niczym kompilatorów z dawnych lat, kiedy to pisało się pojedyncze, monolityczne pliki źródłowe zawierające zbiór procedur. W rozdziale 2. dowiesz się, że klasa jest wzorem, na podstawie którego tworzone są obiekty, oraz że w Javie prawie wszystko to obiekty.

P: Czy metodę main() należy umieszczać we wszystkich tworzonych klasach?

O: Nie. Programy pisane w Javie mogą się składać z dziesiątek (a nawet setek) klas, lecz może w nich istnieć tylko *jedna* metoda main() — ta jedyna, służąca do uruchamiania programu. Niemniej jednak można tworzyć klasy testowe, które posiadają metodę main() i służą do testowania *innych* klas.

P: W języku, którego używałem wcześniej, mogłem tworzyć testy logiczne, wykorzystując przy tym zmienne całkowite. Czy w Javie można napisać coś takiego:

```
int x = 1;  
while (x) { }
```

O: Nie. W Javie *liczby całkowite* oraz wartości *logiczne* nie są typami danych zgodnymi ze sobą. Ponieważ wynik testu logicznego *musi* być wartością logiczną, zatem jedynymi zmiennymi, jakie można sprawdzać bezpośrednio (czyli bez wykorzystania operatora porównania), są *zmienne logiczne*. Na przykład, można napisać:

```
boolean czyGorace = true;  
while (czyGorace) { }
```

Przykłady pętli while

```
public class Petelki {  
    public static void main(String[] args) {  
        int x = 1;  
        System.out.println("Przed wykonaniem pętli");  
        while (x < 4) {  
            System.out.println("Wewnątrz pętli");  
            System.out.println("Wartość x = " + x);  
            x = x + 1;  
        }  
        System.out.println("I już po pętli...");  
    }  
}
```

```
> java Petelki  
Przed wykonaniem pętli  
Wewnątrz pętli  
Wartość x = 1  
Wewnątrz pętli  
Wartość x = 2  
Wewnątrz pętli  
Wartość x = 3  
I już po pętli...
```

a oto wyniki



CELNE SPOSTRZEŻENIA

- Instrukcje muszą kończyć się średnikiem ;.
- Bloki kodu są wyznaczane przez pary nawiasów klamrowych {}.
- Zmienną całkowitą należy definiować, podając jej typ i nazwę, na przykład: int x;.
- Operatorem **przypisania** jest pojedynczy znak równości =.
- Operatorem **równości** są dwa znaki równości ==.
- Pętla while wykonuje wszystkie instrukcje umieszczone wewnątrz jej bloku (wyznaczonego przez nawiasy klamrowe) tak długo, jak długo podany warunek ma wartość *true*.
- Jeśli warunek przyjmie wartość **false**, to kod umieszczony wewnątrz pętli nie zostanie wykonany, a realizacja programu będzie kontynuowana od wiersza znajdującego się bezpośrednio za blokiem pętli.
- Testy logiczne należy umieszczać w nawiasach:
`while (x == 4) { }`

Rozgałęzienia warunkowe

W Javie test `if` to praktycznie to samo co test warunku używany w pętli `while`, z tą różnicą, iż zamiast stwierdzenia „*dopóki* ciągle mamy browara...” stwierdzamy: „*jeśli* ciągle mamy browara...”.

```
class TestIf {
    public static void main(String[] args) {
        int x = 3;
        if (x == 3) {
            System.out.println("x musi mieć wartość 3");
        }
        System.out.println("Ta instrukcja zawsze zostanie wykonana");
    }
}
> java TestIf
x musi mieć wartość 3
Ta instrukcja zawsze zostanie wykonana
```

Wyniki wykonania programu

W powyższym programie wiersz wyświetlający łańcuch znaków „*x musi mieć wartość 3*” jest wykonywany wyłącznie w przypadku, gdy warunek (*x* jest równe 3) będzie spełniony. Z kolei instrukcja wyświetlająca tekst „*Ta instrukcja zawsze zostanie wykonana*” jest wykonywana zupełnie niezależnie od wyniku wcześniejszego warunku. A zatem w zależności od wartości zmiennej *x* może zostać wykonana jedna lub dwie instrukcje wyświetlające komunikaty na ekranie.

Można jednak dodać do warunku klauzulę `else` i dzięki temu powiedzieć coś w stylu: „*Jeśli* wciąż jest browar, to koduj dalej, natomiast w przeciwnym razie kup browar i koduj dalej...”.

```
class TestIf2 {
    public static void main(String[] args) {
        int x = 2;
        if (x == 3) {
            System.out.println("x musi mieć wartość 3");
        } else {
            System.out.println("x jest RÓŻNE od 3!");
        }
        System.out.println("Ta instrukcja zawsze zostanie wykonana");
    }
}
> java TestIf2
x jest RÓŻNE od 3!
Ta instrukcja zawsze zostanie wykonana
```

Wyniki wykonania nowego programu

`System.out.print` kontra

`System.out.println`

Jeśli byłbyś uważny (w co nie wątpimy), na pewno zauważysz, że czasami w programach przykładowych używana była metoda `print`, a czasami `println`.

Czym one się od siebie różnią?

Metoda `System.out.println` dodaje na końcu wyświetlanego łańcucha znak nowego wiersza (nazwę tej metody można potraktować jako `printnewline`, czyli — wydrukuj z nowym wierszem), natomiast kolejne wywołania metody `System.out.print` będą wyświetlać łańcuchy znaków w tym samym wierszu. Jeśli zatem wszystkie wyświetlane łańcuchy znaków mają się znaleźć w osobnych wierszach, to należy wykorzystać metodę `println`. Jeśli natomiast łańcuchy mają być wyświetlane razem, należy się posłużyć metodą `print`.



Zaostre ołówek –

Na podstawie poniższych wyników wyświetlonych na ekranie:

`> java Test
DooBeeDooBeeDo`

uzupełnij brakujące fragmenty kodu:

```
class DooBee {
    public static void main(String[] args) {
        int x = 1;
        while (x < ____) {
            System.out.println("Doo");
            System.out.println("Bee");
            x = x + 1;
        }
        if (x ==____) {
            System.out.println("Do");
        }
    }
}
```

Tworzenie poważnej aplikacji biznesowej

Wykorzystajmy całą zdobytą wiedzę o Javie w słusznym celu i stwórzmy coś praktycznego. Potrzebujemy klasy posiadającej metodę `main()`, zmiennych typu `int` oraz `String`, pętli `while` oraz testu `if`. Jeszcze trochę pracy, a będziesz błyskawicznie tworzyć aplikacje biznesowe działające na serwerach. Jednak *zanim* spojrzyz na kod zamieszczony na tej stronie, zastanów się przez chwilę, w jaki sposób *samemu* napisałeśbyś znaną piosenkę o 99 butelkach piwa.



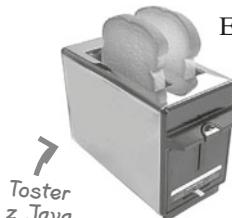
```
class PiosenkaOPiwie {  
    public static void main(String[] args) {  
        int iloscButelek = 99;  
        String slowo = "bottles";  
        while (iloscButelek > 0) {  
            if (iloscButelek == 1) {  
                slowo = "bottle"; // liczba pojedyncza  
            }  
            System.out.println(iloscButelek + " " + slowo + " of beer on the wall");  
            System.out.println(iloscButelek + " " + slowo + " of beer.");  
            System.out.println("Take one down.");  
            System.out.println("Pass it around.");  
            iloscButelek = iloscButelek - 1;  
            if (iloscButelek > 0) {  
                System.out.println(iloscButelek + " " + slowo + " of beer on the wall");  
            } else {  
                System.out.println("No more bottles of beer on the wall");  
            } // koniec else  
        } // koniec while  
    } // koniec metody main  
} // koniec klasy
```

Przedstawiony program wciąż zawiera pewne błędy. Można go poprawnie skompilować i uruchomić, jednak wyniki, jakie generuje, nie są w 100 procentach prawidłowe. Sprawdź, czy potrafisz wykryć błąd i poprawić go.

W poniedziałek rano u Roberta

W poniedziałek rano budzik Roberta dzwoni o 8:30, podobnie jak we wszystkie pozostałe dni tygodnia. Ale Robert ma za sobą szalony weekend i dlatego nacisnął przycisk DRZEMKA — właśnie w tym momencie wszystko się zaczęło i urządzenia obsługiwane przez Javę rozpoczęły działanie.

Najpierw budzik przesyła informację do ekspresu do kawy*: „Hej, mistrzunio ciągle śpi, więc przesuń parzenie kawy o 12 minut”.



Toster z Java

Ekspres do kawy wysyła wiadomość do tostera Motorola™:
„Wstrzymać produkcję tostów, Robert wciąż śpi!”.

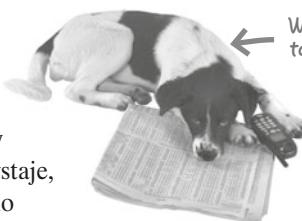
Następnie budzik przesyła do telefonu komórkowego Roberta — Nokia Navigator™ — komunikat następującej treści: „Zadzwoń do Roberta o 9 rano i powiedz mu, że jesteśmy już troszkę spóźnieni”. I, w końcu, budzik przesyła do bezprzewodowej obroży Burka (Burek to pies Roberta), aż nazbyt dobrze znany komunikat: „Lepiej przynieś gazetę — na spacer nie ma co liczyć”.

Kilka minut później alarm w budziku włącza się ponownie, i także tym razem Robert naciska przycisk DRZEMKA, a urządzenia ponownie zaczynają swoje pogawędki. W końcu alarm włącza się po raz trzeci. Lecz teraz, gdy tylko Robert wyciąga rękę do wiadomego przycisku, budzik przesyła do obroży Burka komunikat „skacz i szczekaj”. Obudzony i przerzążony Robert od razu wstaje, dziękując w duchu swej znajomości Javy i niewielkiej wyprawie do sklepu Radio Bajer™, która w tak znaczący sposób usprawniła jego codzienne życie.



Java w budziku

Także tu jest Java



W obroży Burka także jest Java

Tost Roberta jest upieczoney.



tu posmarowano masłem

Jego kawa paruje w filizance.

A gazeta leży na miejscu.

Ot, kolejny poranek w domu obsługiwianym przez Javę.

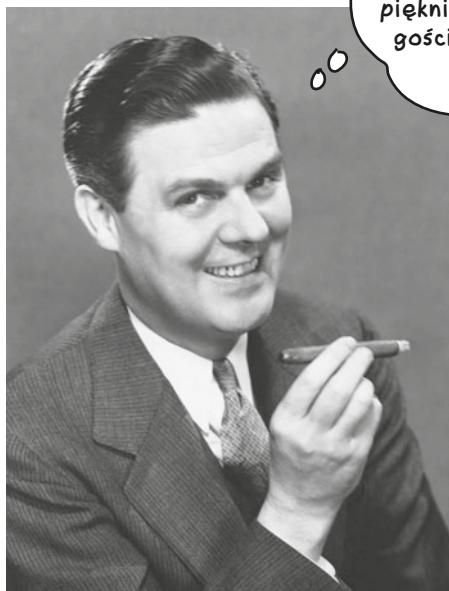
Także Ty możesz mieszkać w takim domu. Korzystaj z rozwiązań bazujących na języku Java, Ethernetie i technologii Jini. Bądź świadom ograniczeń innych platform określanych jako „podłącz i pracuj” (co w rzeczywistości oznacza: podłącz i pracuj kolejne 3 dni nad tym, aby to wszystko uruchomić) lub „przenośne”. Ela, siostra Roberta, wypróbowała jedną z tych *innych* technologii i wyniki nie były, jakby to rzec, ani przekonujące, ani bezpieczne. Lepiej w ogóle nie wspominać, co zrobił jej pies...



Czy ta opowieść mogłaby być prawdziwa? I tak, i nie. Choć istnieją wersje Javy działające w różnego typu urządzeniach, takich jak komputery przenośne PDA, telefony komórkowe (zwłaszcza telefony komórkowe), pagery, karty intelligentne i tak dalej, to jednak trudno by było znaleźć obrożę lub toster obsługiwany przez Javę. Jednak jeśli nawet nie udałoby Ci się znaleźć wersji ulubionego gadżetu obsługiwanej przez Javę, to i tak będziesz w stanie kontrolować go tak, jak gdyby był wyposażony w ten język, wystarczy w tym celu użyć innego interfejsu (na przykład laptopa), w którym można korzystać z Javy. Takie rozwiązanie nosi nazwę *architektury zastępczej* Jini. Tak, faktycznie można mieć taki niesamowity dom.

* Jeśli jesteś ciekaw protokołu, to jest do tego celu wykorzystywane *rozsyłanie grupowe IP*.

Napiszmy program



Wypróbuje mój nowy program „krasomówczy”, a będziesz mówić równie pięknie i mądrze jak szef lub goście z działu marketingu.

```
public class Krasomowca {  
    public static void main (String[] args) {  
  
        1 // trzy grupy słów, które będą wybierane do zdania (dodaj własne!)  
        String[] listaSlow1 = {"architektura wielowarstwowa",  
            "30000 metrów", "rozwiązań B-do-B", "aplikacja kliencka",  
            "interfejs internetowy", "inteligentna karta", "rozwiązań  
            dynamiczne", "sześć sigma", "przenikliwość"};  
  
        String[] listaSlow2 = {"zwiększa możliwości", "poprawia atrakcyjność",  
            "pomaga wartości", "opracowana dla", "bazująca na", "rozproszona",  
            "sieciowa", "skoncentrowana na", "podniesiona na wyższy poziom",  
            "skierowanej", "udostępniona"};  
  
        String[] listaSlow3 = {"procesu", "punktu wpisywania", "rozwiązań",  
            "strategii", "paradygmatu", "portalu", "witryny", "wersji", "misji"};  
  
        2 // określenie, ile jest słów w każdej z list  
        int lista1Dlugosc = listaSlow1.length;  
        int lista2Dlugosc = listaSlow2.length;  
        int lista3Dlugosc = listaSlow3.length;  
  
        3 // generacja trzech losowych słów (lub zwrotów)  
        int rnd1 = (int) (Math.random() * lista1Dlugosc);  
        int rnd2 = (int) (Math.random() * lista2Dlugosc);  
        int rnd3 = (int) (Math.random() * lista3Dlugosc);  
  
        4 // stworzenie zdania  
        String zdanie = listaSlow1[rnd1] + " " + listaSlow2[rnd2] + " " +  
            listaSlow3[rnd3];  
  
        5 // wyświetlenie zdania  
        System.out.println("To jest to, czego nam trzeba: " + zdanie);  
    }  
}
```

Program krasomówczy

Jak to działa?

Najogólniej rzecz biorąc, program tworzy trzy listy słów (zwrotów), następnie losowo wybiera po jednym elemencie z każdej z nich i wyświetla wyniki. Nie przejmuj się, jeśli nie rozumiesz *dokładnie*, jak działa każda z instrukcji programu. W końcu masz przed sobą całą książkę, zatem wyluzuj się. To tylko rozwiązanie B-do-B pomnaża wartość paradygmatu.

1. Pierwszym krokiem jest utworzenie trzech tablic łańcuchów znaków — będą one zawierać wszystkie słowa i zwroty. Deklarowanie i tworzenie tablic jest bardzo łatwe, oto prosta tablica:

```
String[] zwierzaki = {"Burek", "Azorek", "As"};
```

Wszystkie słowa są zapisane w cudzysłowach (dotyczy to wszystkich grzeczych łańcuchów znaków) i oddzielone od siebie przecinkami.

2. Naszym celem jest losowe wybranie z każdej z list (tablic) jednego słowa, a zatem musimy wiedzieć, jakie są ich długości. Jeśli na liście jest 14 słów, to potrzebna nam będzie liczba losowa z zakresu od 0 do 13 (w Javie pierwszy element tablicy ma indeks 0, a zatem pierwsze słowo znajduje się w komórce tablicy o numerze 0, drugie — w komórce o numerze 1, a ostatnie, czternaste słowo, w komórce o numerze 13). Na szczęście tablice w Javie całkiem chętnie informują nas o swojej długości, co jest bardzo wygodne. Kontynuując przykład tablicy z imionami ulubieńców, można zapytać:

```
int x = zwierzaki.length;
```

a w zmiennej **x** zostanie zapisana wartość 3.

3. Teraz potrzeba nam trzech liczb losowych. Java ma w standardzie na wyposażeniu wbudowaną profesjonalną i kompetentną grupę metod matematycznych (jak na razie możesz je sobie wyobrażać jako zwyczajne funkcje). Metoda **random()** zwraca wartość z zakresu od 0 do „prawie” 1, a zatem musimy pomnożyć tę wartość przez ilość elementów używanej listy (czyli przez długość tablicy). Musimy też wymusić, aby uzyskany wynik był liczbą całkowitą (miejscza po przecinku nie są dozwolone!) i dlatego używamy rzutowania typów (szczegółowe informacje na ten temat zostaną podane w rozdziale 4.). Niczym się to nie różni od sytuacji, w której trzeba skonwertować liczbę zmiennoprzecinkową na całkowitą:

```
int x = (int) 24.5;
```

4. Teraz musimy skonstruować zdanie, wybierając po jednym elemencie z każdej z trzech list i łącząc je w jedną całość (jednocześnie umieszczamy pomiędzy nimi znaki odstępu). Należy w tym celu użyć operatora **+**, który łączy (osobiście wolimy bardziej techniczne określenie „*przeprowadza konkatenację*”) obiekty String. Aby pobrać element z tablicy, należy podać jego numer jako indeks (czyli pozycję):

```
String s = zwierzaki[0]; // s ma wartość "Burek"
s = s + " " + "to pies"; // s ma wartość "Burek to pies"
```

5. W końcu wyświetlamy utworzony łańcuch znaków w strumieniu wyjściowym. I... voila! *W ten sposób przechodzimy do działu marketingu.*

Oto co my tutaj mamy:

architektura wielowarstwowa pomnaża wartość misji

przenikliwość skierowanej strategii

inteligentna karta podniesiona na wyższy poziom paradygmatu

rozwiązanie B-do-B poprawia atrakcyjność portalu

aplikacja kliencka opracowana dla rozwiązania

Kompilator i wirtualna maszyna Javy (JVM)

Pogawędki przy kominku



Virtualna maszyna Javy

Co? Chyba sobie żartujesz! **Halo!** To ja jestem Javą. To dzięki mnie można wykonywać programy. Kompilator jedynie tworzy plik. I to wszystko. Nic więcej — tylko plik. Możesz go sobie wydrukować, użyć jako tapety, wyłożyć nim klatkę dla ptaków lub zrobić z nim cokolwiek innego — plik będzie bezużyteczny, dopóki ktoś nie poprosi się mnie o jego wykonanie.

No i właśnie... kolejna sprawa... kompilator nie ma za grosz poczucia humoru. Jeszcze raz powtórz, że jeśli miałbyś cały dzień poświęcić na sprawdzanie tych malutkich i głupich błędów składniowych...

No przecież nie mówię, że jesteś całkowicie bezużyteczny. Ale czym ty się w zasadzie zajmujesz? Tak naprawdę... Bo w zasadzie to nie wiem. Programiści mogliby w gruncie rzeczy sami pisać kod bajtowy, a ja mogłabym je wykonywać. I szybciutko straciłbym pracę, bratku.

Pominę humorystyczne aspekty twojej wypowiedzi, ale wciąż nie odpowiedziałeś na moje pytanie. Co ty w zasadzie robisz?

Tematem dzisiejszej pogawędki jest spór pomiędzy kompilatorem a wirtualną maszyną Javy na temat zagadnienia: „**Które z tych narzędzi jest ważniejsze?**”.

Kompilator

Nie podoba mi się ten ton.

Wypraszam sobie, ale gdyby nie ja, to w zasadzie co byś była w stanie wykonywać? Dla twojej informacji — jest pewien powód, dla którego Java została zaprojektowana tak, aby wykonywała kod bajtowy. Gdyby Java została zaprojektowana jako język całkowicie interpretowany, w którym wirtualna maszyna musi w czasie rzeczywistym interpretować kod źródłowy prosto z edytora tekstów, to programy Javy działałyby w tempie kulawego lodowca. Na szczęście Java miała wystarczająco dużo czasu, aby przekonać programistów, że w końcu jest wystarczająco szybka i potężna, aby można ją było stosować do realizacji większości zadań.

Wybacz, ale wychodzi tu twoja ignorancja (nie żebym mówił, że jesteś arogancka). Choć to prawda — teoretycznie rzecz biorąc — że mogłabyś wykonywać wszystkie poprawnie sformatowane kody bajtowe, nawet gdyby nie zostały utworzone przez kompilator, to w praktyce jest to założenie absurdalne. Ręczne pisanie kodów bajtowych to jak gdyby edycja tekstu poprzez przesuwanie poszczególnych pikseli na ekranie monitora. I byłbym wdzięczny, gdybyś nie zwracała się do mnie w ten sposób.

Virtualna maszyna Javy

Ale niektóre wciąż do mnie docierają! Mogę zgłosić wyjątek `ClassCastException`, a czasami programiści próbują zapisywać dane w tablicach, które zostały zadeklarowane do przechowywania czegoś innego, i...

OK. Pewnie. Ale co z *bezpieczeństwem*? Spójrz na te wszystkie metody zabezpieczeń, które ja wykonuję... A ty co? Sprawdzasz, czy ktoś nie zapomniał wpisać średnika? Och, to naprawdę poważne zagrożenie bezpieczeństwa! Dzięki ci za to o kompilatorze!

Nieważne. Ja też muszę wykonywać tę samą pracę tylko po to, aby przed wykonaniem programu upewnić się, czy ktoś nie zmieniał kodów bajtowych, które ty wcześniej skompilowałeś.

Och, możesz na to liczyć. *Bratku*.

Kompilator

Pamiętasz zapewne, że Java jest językiem o ścisłej kontroli typów, co oznacza, że nie mogę pozwolić, aby zmienne zawierały dane niewłaściwych typów. To kluczowe zagadnienie bezpieczeństwa i to ja jestem w stanie zapobiec pojawianiu się znacznej większości nieprawidłowości, zanim w ogóle będą mogły przeszkodzić ci w działaniu. Poza tym...

Przepraszam, ale jeszcze nie skończyłem. Owszem, *istnieją* pewne wyjątki związane z typami, które mogą się pojawiać w czasie działania programu, ale trzeba na nie pozwolić ze względu na kolejną niezwykle ważną cechę Javy, a mianowicie — dynamiczne wiązanie. W czasie działania programów pisanych w Javie można do nich dołączać nowe obiekty, o których programista tworzący program nawet *nie wiedział*, dlatego też muszę zezwolić na pewną elastyczność. Jednak moje zadanie polega na wykryciu kodu, który spowoduje — *mogłby* spowodować — wystąpienie błędów podczas działania programu. Zazwyczaj jestem w stanie określić, kiedy coś nie będzie działać, na przykład kiedy programista użyje wartości logicznej zamiast połączenia sieciowego, wykryję to i zabezpieczę go przed problemami, jakie pojawiłyby się po uruchomieniu programu.

Przepraszam bardzo, ale jestem pierwszą linią obrony, jak zwykło się mówić. Naruszenia typów danych, o jakich wcześniej mówiłem, mogłyby doprowadzić do prawdziwego chaosu w programie, gdyby tylko się pojawiły. Dodatkowo wykrywam i zapobiegam naruszeniom praw dostępu, takim jak próba wywołania metody prywatnej lub zmiana zmiennych, które ze względów bezpieczeństwa, nigdy nie powinny być zmieniane. Uniemożliwiam programistom przeprowadzanie modyfikacji kodu, którego nie powinni zmieniać, w tym także kodu, który próbuje uzyskać dostęp do krytycznych informacji innych klas. Opisanie znaczenia mojej pracy mogłoby zająć wiele godzin, a może nawet i dni.

Oczywiście, ale jak już wcześniej zauważyłem, gdybym ja nie zapobiegał około 99 procentom potencjalnych problemów, to ty bez wątpienia szybko byś się „zawiesiła”. Poza tym, wygląda na to, że nie masz już czasu, a zatem wróćmy do tego tematu w kolejnej pogawędce.

Ćwiczenia: magnesiki z kodem



Magnesiki z kodem

Działający program Javy został podzielony na fragmenty. Czy jesteś w stanie złożyć go z powrotem w jedną całość, tak aby wygenerował przedstawione poniżej wyniki? Niektóre nawiasy klamrowe spadły na podłogę i były zbyt małe, aby można je było podnieść, dlatego w razie potrzeby możesz je dodawać!

```
if (x == 1) {  
    System.out.print("d");  
    x = x - 1;  
}
```

```
if (x == 2) {  
    System.out.print("b c");  
}
```

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

```
{ if (x > 2) {  
    System.out.print("a");  
}
```

```
int x = 3;
```

```
x = x - 1;  
System.out.print("-");
```

```
while (x > 0) {
```

Wyniki:

```
Wiersz polecenia  
T:\r01>java Układanka  
a-b c-d
```



Ćwiczenie



BĄDŹ kompilatorem

Każdy z plików przedstawionych na tej stronie to niezależny, kompletny plik źródłowy Javy. Twoim zadaniem jest wcieLENIE się w kompilator i określenie, czy poszczególne pliki można poprawnie skompilować. Jeśli plików nie można skompilować, to jak należy je poprawić? Z kolei, jeśli można je skompilować, to jakie będą wyniki ich działania?

A

```
class CwiczenielA {
    public static void main(String[] args) {
        int x = 1;
        while (x < 10) {
            if (x > 3) {
                System.out.println("Wielkie X");
            }
        }
    }
}
```

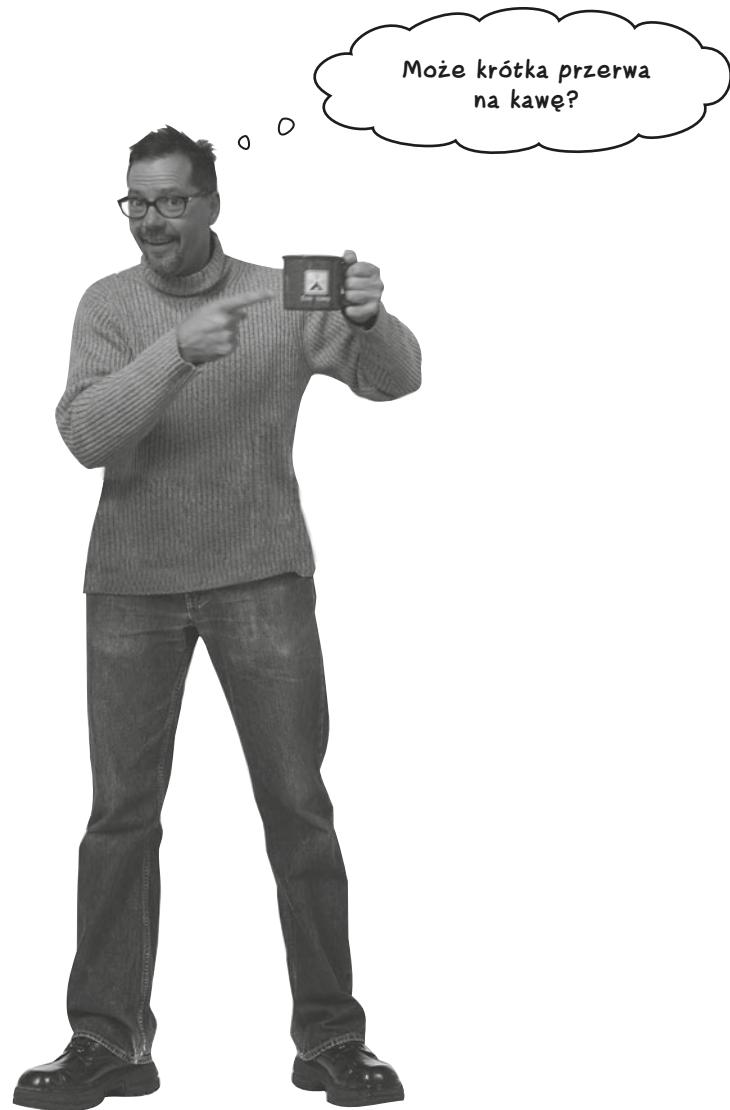
B

```
public static void main() {String[] args) {
    int x = 5;
    while (x > 1) {
        x = x - 1;
        if (x < 3) {
            System.out.println("Malutkie x");
        }
    }
}
```

C

```
class CwiczenielC {
    int x = 5;
    while (x > 1) {
        x = x - 1;
        if (x < 3) {
            System.out.println("Malutkie x");
        }
    }
}
```

Przerwa na kawę





Pomieszczone komunikaty

Poniżej zamieszczono prosty program napisany w Javie. Brakuje w nim jednego fragmentu. Twoim zadaniem jest **dopasowanie proponowanych bloków kodu** (przedstawionych w lewej kolumnie) z **wynikami**, które program wygeneruje po wstawieniu wybranego bloku. Nie wszystkie wiersze wyników zostaną wykorzystane, a niektóre z nich mogą być wykorzystane więcej niż jeden raz. Narysuj linie łączące bloki kodu z odpowiadającymi im wynikami. (Wszystkie odpowiedzi mozna znaleźć na końcu rozdziału).

```
class Test {
    public static void main(String [] args) {
        int x = 0;
        int y = 0;
        while ( x < 5 ) {
            
            System.out.print(x + " " + y + " ");
            x = x + 1;
        }
    }
}
```

Tutaj należy umieścić wybrany blok kodu



Potocz proponowane bloki kodu z generowanymi przez nie wynikami

Bloki kodu:

y = x - y;

y = y + x;

```
y = y + 2;
if( y > 4 ) {
    y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
    x = x + 1;
    if ( y < 3 ) {
        x = x - 1;
    }
}
y = y + 2;
```

Generowane wyniki:

22 46

11 34 59

02 14 26 38

02 14 36 48

00 11 21 32 42

11 21 32 42 53

00 11 23 36 410

02 14 25 36 47

Zagadka: Zagadkowy basen



Zagadkowy basen

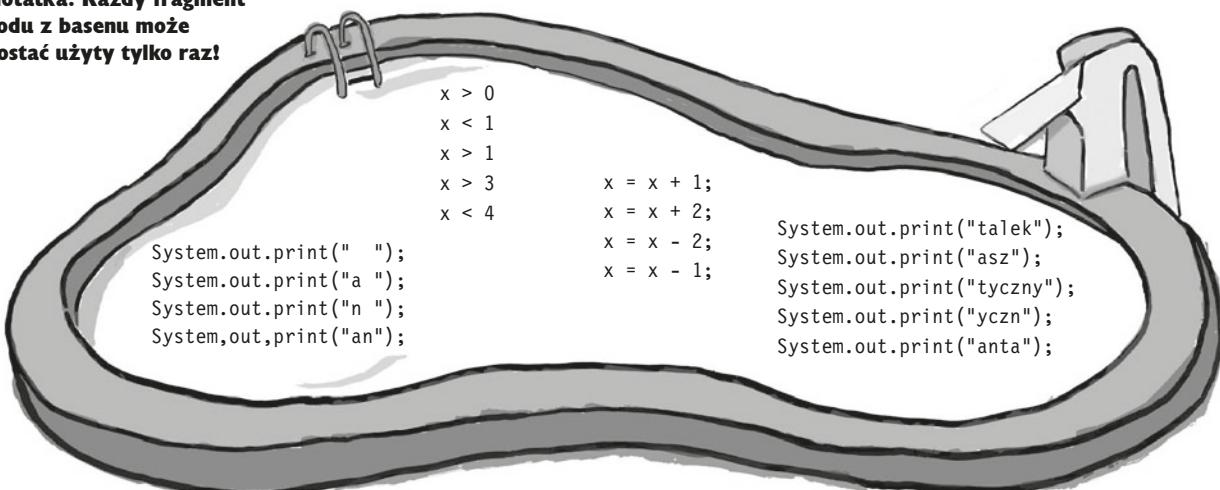
Twoim **zadaniem** jest wybranie fragmentów kodu z basenu i umieszczenie ich w miejscach kodu oznaczonych podkreśleniami. żadnego fragmentu kodu **nie można** użyć więcej niż raz, a co więcej, nie wszystkie fragmenty zostaną wykorzystane. **Zadanie** polega na stworzeniu klasy, którą będzie można skompilować i która wygeneruje wyniki przedstawione poniżej. Nie daj się zwieść pozorom — ta zagadka jest trudniejsza, niż można by przypuszczać.

Wyniki:

```
cz. Wiersz polecenia  
T : \r@1>java ZagadkowyBasen1  
a nasz  
antalek  
antyczny  
T : \r@1>
```

```
class ZagadkowyBasen1 {  
    public static void main(String [] args) {  
        int x = 0;  
  
        while ( _____ ) {  
  
            if ( x < 1 ) {  
                _____  
            }  
  
            if ( _____ ) {  
                _____  
            }  
            if ( x == 1 ) {  
  
                if ( _____ ) {  
                    _____  
                }  
                System.out.println("");  
  
            }  
        }  
    }  
}
```

Notatka: Każdy fragment kodu z basenu może zostać użyty tylko raz!





Ćwiczenie rozwiążanie

Magnesiki z kodem

```
class Ukladanka {
    public static void main(String[] args) {
        int x = 3;
        while (x > 0) {
            if (x > 2) {
                System.out.print("a");
            }
            x = x - 1;
            System.out.print("-");
            if (x == 2) {
                System.out.print("b c");
            }
            if (x == 1) {
                System.out.print("d");
                x = x - 1;
            }
        }
    }
}
```

Wiersz polecenia

```
T:\r01>java Ukladanka
a-b c-d
```

```
A class Cwiczenie1A {
    public static void main(String[] args) {
        int x = 1;
        while (x < 10) {
            x = x + 1;
            if (x > 3) {
                System.out.println("Wielkie X");
            }
        }
    }
}
```

W oryginalnej postaci program można uruchomić i skompilować, jednak bez dodanego wiersza kodu program wpadnie w nieskończoną pętlę!

```
B class Cwiczenie1B {
    public static void main(String[] args) {
        int x = 5;
        while (x > 1) {
            x = x - 1;
            if (x < 3) {
                System.out.println("Malutkie x");
            }
        }
    }
}
```

Tego pliku nie da się skompilować bez dodania deklaracji klasy. Dodatkowo **nie można zapomnieć** o dodaniu zamykającego nawiasu klamrowego!

```
C class Cwiczenie1C {
    public static void main(String[] args) {
        int x = 5;
        while (x > 1) {
            x = x - 1;
            if (x < 3) {
                System.out.println("Malutkie x");
            }
        }
    }
}
```

Kod pętli while musi być umieszczony wewnętrz metody. Nie można umieszczać go bezpośrednio w kodzie klasy.

Odpowiedzi na zagadki

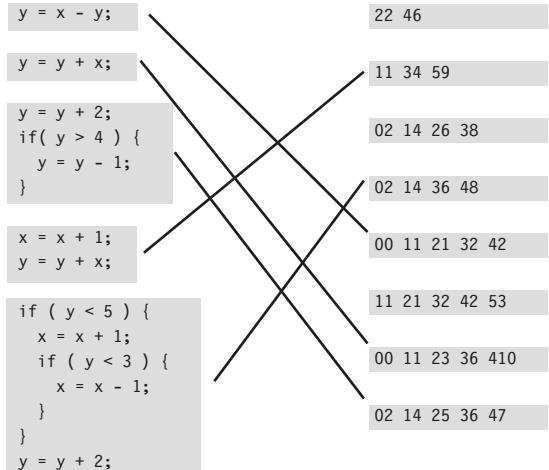


```
class ZagadkowyBasen1 {  
    public static void main(String[] args) {  
        int x = 0;  
        while ( x < 4 ) {  
            System.out.print("a");  
            if ( x < 1) {  
                System.out.print(" ");  
            }  
            System.out.print("n");  
            if ( x > 1 ) {  
                System.out.print("tyczny");  
                x = x + 2;  
            }  
            if ( x == 1 ) {  
                System.out.print("talek");  
            }  
            if ( x < 1 ) {  
                System.out.print("asz");  
            }  
            System.out.println("");  
            x = x + 1;  
        }  
    }  
}
```

```
Wiersz polecenia  
T:\r01>java ZagadkowyBasen1  
a nasz  
antalek  
antyczny  
T:\r01>
```

```
class Test {  
    public static void main(String [] args) {  
        int x = 0;  
        int y = 0;  
        while ( x < 5 ) {  
            System.out.print(x + " " + y + " ");  
            x = x + 1;  
        }  
    }  
}
```

Bloki kodu:



2. Klasy i obiekty

Wycieczka do Obiektowa



Mówiono mi, że będą obiekty. W rozdziale 1. cały tworzony kod był umieszczany w metodzie `main()`. Nie jest to poprawne rozwiązanie obiektowe. W rzeczywistości, z punktu widzenia programowania zorientowanego obiektowo, jest to rozwiązanie *całkowicie* niewłaściwe. Cóż, w programie „krasomówczym” wykorzystaliśmy *kilka* obiektów, takich jak tablice łańcuchów znaków (obiektów `String`), jednak nie stworzyliśmy żadnego własnego *typu obiektowego*. Zatem teraz musimy zostawić świat programowania proceduralnego, usunąć to co najważniejsze z metody `main()` i zacząć tworzyć własne obiekty. Przyjrzymy się czynnikom, które sprawiają, że programowanie zorientowane obiektowo przy użyciu języka Java, jest takie fajne. Przedstawimy różnice pomiędzy *klasą* a *obiektem*. Przekonamy się, w jaki sposób obiekty mogą ułatwić nam życie (a przynajmniej jego aspekty związane z programowaniem; nie będziemy bowiem w stanie sprawić, abyś zaczął się znać na modzie i wyrobił sobie dobry gust). Jedno ostrzeżenie: kiedy już dotrzesz do Obiektowa, możesz już nigdy się z niego nie wydostać. Wyślij nam pocztówkę.

Wojna o fotel (albo Jak Obiekty Mogą Zmienić Twoje Życie)

Dawno temu w sklepie z oprogramowaniem dwóch programistów dostało tę samą specyfikację i kazano im „napisać co trzeba”.

Naprawdę Denerwujący Szef Projektu zmusił obu programistów do rywalizowania z sobą, obiecując im, że ten, który pierwszy odda kod, dostanie Superfotel™, który mają wszyscy programiści w Dolinie Krzemowej. Bronek — programista proceduralny — oraz Jurek — programista obiektowy — wiedzą, że wykonanie zadania będzie jak przysłowiowa „kaszka z mleczkiem”.

Bronek, siedząc w swoim boksie, pomyślał: „Jakie rzeczy ten program ma robić? Jakich **procedur** potrzebuje?”. Po czym sam sobie odpowiedział: „Potrzebne mi będą procedury: **obróć** i **odtwarzDźwięk**”. I zabrał się za pisanie odpowiednich procedur. No bo w końcu czym innym jest program, jeśli nie zbiorem stosownych procedur?

W międzyczasie Jurek poszedł do kawiarni na filiżankę kawy i zadał sobie pytanie: „Jakie są **obiekty** w tym programie... jacy są jego najważniejsi bohaterowie?”. Pierwszą odpowiedzią, jaka mu przyszła do głowy, była:

Figury. Oczywiście w programie występują też inne obiekty, takie jak Użytkownik, Dźwięk czy też zdarzenie Kliknięcie, ale Jurek dysponuje już biblioteką obsługującą te obiekty, zatem może się skoncentrować na tworzeniu Figur. Przeczytaj dalszą treść rozdziału, aby się przekonać, jak Jurek i Bronek tworzyli swoje programy, a przede wszystkim, aby uzyskać odpowiedź na najbardziej palące pytanie: „**Kto dostanie Superfotel?**”.

W boksie Bronka

Jak miliardy razy wcześniej Bronek zabrał się do pisania **Niezwykle Ważnych Procedur**. Błyskawicznie napisał procedury **obroc** oraz **odtwarzDźwięk**:

```
obroc(numFigury) {  
    // obrócenie figury o 360 stopni  
}  
  
odtwarzDźwięk(numFigury) {  
    // na podstawie numeru figury określ,  
    // jaki plik AIF należy odtworzyć i odtwórz go  
}
```

Na graficznym interfejsie użytkownika programu będą wyświetlane figury: kwadrat, okrąg oraz trójkąt. Gdy użytkownik kliknie kształt, zostanie on obrócony o 360 stopni zgodnie z ruchem wskaźówek zegara, po czym program odtworzy plik dźwiękowy zapisany w formacie AIF, odpowiadający klikniętej figurze.



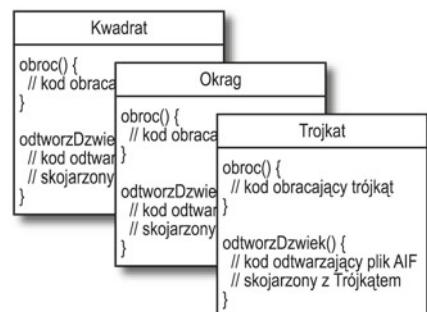
specyfikacja



fotel

W kawiarni na laptopie Jurka

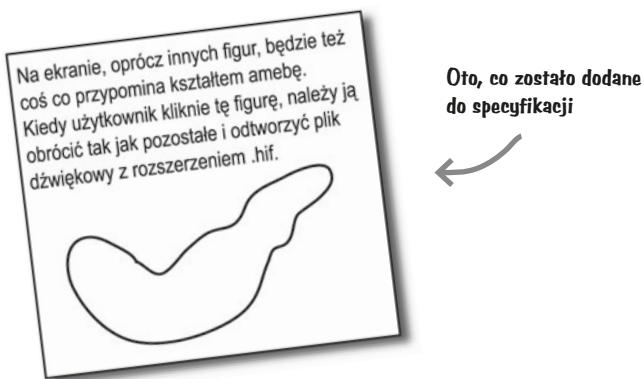
Jurek stworzył **klasy** dla każdej z trzech figur.



**Bronek myślał, że już ma w garści wygraną.
Już niemal mógł czuć miękką skórę Superfotela pod swoim...**

Ale chwileczkę! Nastąpiła zmiana specyfikacji.

- W porządku. *Technicznie* rzecz biorąc, wygrał Bronek — powiedział Szef — ale musimy dodać do programu jedną malutką rzeczą. Dla takich doświadczonych programistów jak wy, nie będzie to stanowiło żadnego problemu.
- *Gdyby dostawać dziesięć groszy za każdą zmianę specyfikacji...* — pomyślał Bronek.
- *A jednak Jurek jest dziwnie spokojny. Co jest grane?* Jednak Bronek wciąż trzymał się swojej opinii, że podejście obiektowe, choć ciekawe, jest wolne. Pomyślał też, że jeśli ktoś chciałby sprawić, by zmienił zdanie, musiałby poddać go bezlitosnym torturom.



Z powrotem w boksie Bronka

Procedura obracania nawet po zmianach będzie działać dobrze, podobnie jak kod służący do przejrzenia tablicy figur i dopasowania wartości numFigury do faktycznej figury. **Ale trzeba zmienić procedurę odtworzDzwiek.** I co to jest ten cały plik *.hif*?

```
odtworzDzwiek(numFigury) {
    // jeśli kształt to nie "ameba",
    // to na podstawie numeru figury określ,
    // jaki plik AIF należy odtworzyć i odtwórz
    // go
    // w przeciwnym razie
    // odtwórz plik dźwiękowy .hif skojarzony
    // z "amebą"
}
```

Okazało się, że zmiany nie są takie straszne, *niemniej jednak konieczność modyfikowania już sprawdzonego kodu spowodowała u Bronka pewne uczucie niepokoju*. W końcu właśnie on, jak nikt inny, powinien doskonale wiedzieć, że niezależnie od tego co mówi szef projektu, *specyfikacja zawsze się zmienia*.

Na plaży na laptopie Jurka

Jurek, uśmiechnął się, wychylił łyżczek swojej Margherity i napisał jedną nową klasę. Czasami sobie myślał, że rzeczą, którą najbardziej kocha w programowaniu obiektowym, jest brak konieczności modyfikowania kodu, który już raz został przetestowany i udostępniony.

Elastyczność, rozszerzalność... — mruczał pod nosem Jurek, przypominając sobie wszystkie zalety programowania obiektowego.

Ameba
obroc() { // kod obracający "amebę" } odtworzDzwiek() { // kod odtwarzający plik .hif // skojarzony z Amebą }

Bronek wpadł do biura Szefa tuż przed Jurkiem

(Acha! I tyle są warte te wszystkie obiektowe bzdury). Ale uśmiech na jego twarzy szybko zgasł, kiedy Naprawdę Denerwujący Szef Projektu powiedział: „Ależ nie! Ameba miała być obracana w zupełnie *inny* sposób...”.

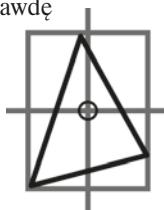
Okazuje się, że kod obracający figury napisany przez obu programistów działa w następujący sposób:

1) Określa prostokątny obszar, w jakim mieści się figura.

2) Wyznacza środek tego obszaru i obraca go wokół tego punktu.

Ale, jak się okazuje, kształt przypominający amebę miał być obracany wokół punktu znajdującego się na jego *końcu*, zupełnie tak samo jak wskazówka zegara.

— Jestem ugotowany — pomyślał Bronek, wyobrażając sobie bulgoczący KociołekSzamana™. — Chociaż, niech pomyślę... Mógłbym przecież dodać do procedury obracającej figury jeszcze jedną instrukcję `if-else` i w niej zakodować na stałe punkt obrotu dla ameby. To prawdopodobnie nie zepsułoby całej procedury. Ale wtedy cichutki głosik w jego głowie odezwał się: — *Wielki błąd. Czy jesteś absolutnie pewny, że specyfikacja znowu się nie zmieni?*



O tym beztrøsco zapomniano
napisać w specyfikacji

Z powrotem w boksie Bronka

Bronek doszedł do wniosku, że lepszym rozwiązaniem będzie dodanie do procedury obracającej figury argumentów, które określają współrzędne punktu obrotu. **Wprowadził to poważne zmiany w kodzie.** Testowanie, rekompilacja, cała masa roboty, którą trzeba wykonać od nowa. To, co wcześniej działało, teraz przestało działać.

```
obroc(numFigury, xPt, yPt) {  
    // jeśli figura to nie "ameba",  
    // wyznaczenie środka na podstawie  
    // prostokąta opisanego  
    // i obrócenie figury o 360 stopni  
    // w przeciwnym razie  
    // wyznaczenie punktu obrotu z uwzględnieniem  
    // podanego przesunięcia xPt, yPt i  
    // obrócenie figury o 360 stopni  
}
```

Na laptopie Jurka, gdzieś na widowni Festiwalu Kapel Wirtualnych

Nie tracąc nawet jednego taktu z prezentowanych utworów, Jurek zmodyfikował **metodę** obracającą, ale wyłącznie w klasie Ameba. **W żaden sposób nie zmienił żadnego z przetestowanych, skompilowanych i działających kodów** stanowiących pozostałe części programu. Aby określić punkt obrotu figury przypominającej amebę, Jurek dodał **atrybuty**, które będą mieć wszystkie „ameby”. Zmodyfikował, przetestował i przesłał (oczywiście bezprzewodowo) zmodyfikowaną wersję programu w czasie trwania utworu *Serce metody*.

Ameba
int xPO int yPO obroc() { // kod obracający amebę // wokół punktu xPO, yPO } odtworzDzwiek() { // kod do odtwarzania nowych // plików dźwiękowych .hif // używanych przez ameby }

Czyli to Jurek — „obiektowiec” — zdobył Superfotel, czy tak?

Nie tak szybko. Bronek znalazł pewną wadę rozwiązania przedstawionego przez Jurka. A ponieważ uważało, że jeśli zdobędzie Superfotel, to zdobędzie także panią Lucynkę z księgowości, musiał zatem przystąpić do natarcia.

BRONEK: W twoim programie powtarzają się te same fragmenty kodu! Procedura do obracania jest we wszystkich tych rzeczach... no — figurach.

JUREK: To jest *metoda*, a nie *procedura*. A to nie są rzeczy tylko *klasy*.

BRONEK: Nieważne. Idiotyczny projekt. Musisz mieć aż *cztery* różne „metody” do obracania. Czy taki projekt może być dobry?

JUREK: O! Jak mniemam, nie widziałeś ostatecznej wersji. Pozwól, że ci pokaże coś, co w programowaniu obiektowym określamy jako **dziedziczenie**.

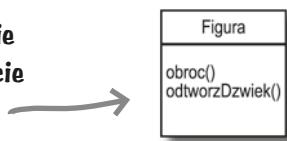


To o niej marzył Bronek (zakładał, że Superfotel zrobi odpowiednie wrażenie)



2

To wszystko są Figury, wszystkie je można obracać, a ich kliknięcie powoduje obrót i odźwiękowanie pliku dźwiękowego. Dlatego też wyodrębniałem ich wspólne cechy i umieściłem je w nowej klasie o nazwie Figura.



Należy to rozumieć jako: „Kwadrat dziedziczy po Figurze”, „Okrąg dziedziczy po Figurze” i tak dalej. Usunąłem metody obroc() i odtworzDzwiek() z pozostałych klas, dzięki czemu teraz jest tylko jedna kopia tej metody.

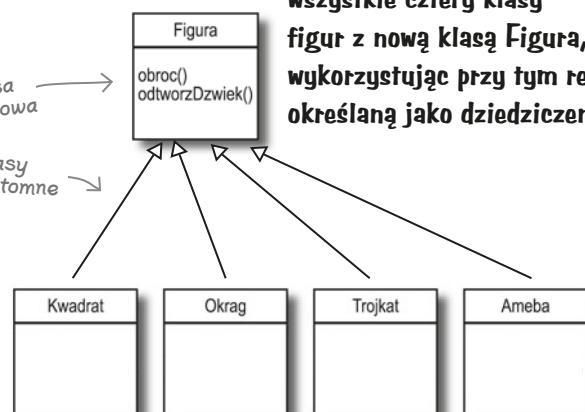
Klasa Figura jest nazywana klasą **bazową** dla pozostałych czterech klas. Natomiast te cztery klasy, są klasami **potomnymi** klasy Figura. Klasa potomna dziedziczy metody klasy bazowej. Innymi słowy, jeśli klasa Figura ma jakieś możliwości funkcjonalne, to możliwości te są automatycznie dostępne w klasach potomnych.

1

Wybrałem elementy, które są wspólne dla wszystkich czterech klas.

3

Następnie połączylem wszystkie cztery klasy figur z nową klasą Figura, wykorzystując przy tym relację określającą jako dziedziczenie.



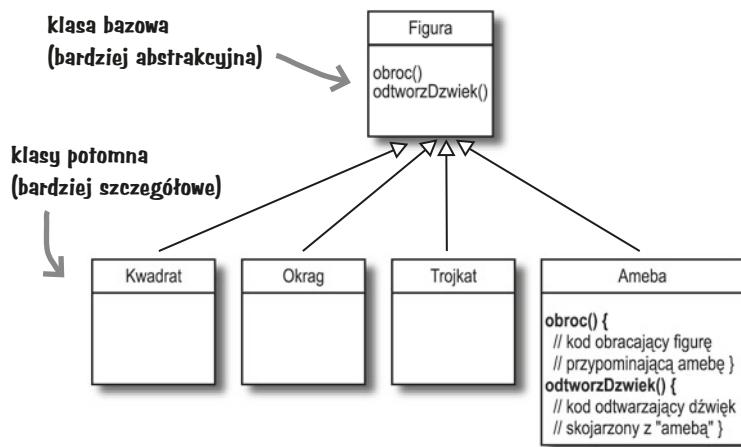
A co z metodą obroc() dla „ameby”?

BRONEK: Ale na czym polegał cały problem — czy nie na tym, że figura przypominająca kształtem amebę miała mieć całkowicie inne procedury obroc() i odtworzDzwiek()?

JUREK: Metody.

BRONEK: Nieważne. W jaki sposób „ameba” może robić coś innego, jeśli dziedziczy możliwości funkcyjne po klasie Figura?

JUREK: To ostatni etap zadania. Klasa Ameba może prześlonić metody klasy Figura. Następnie, podczas wykonywania programu, kiedy każemy obrócić „amebę”, JVM będzie dokładnie wiedzieć, jaką metodę obroc() należy wykonać.



4

W klasie Ameba prześlamiam metody obroc() i odtworzDzwiek() bazowej klasy Figura.

Prześlanianie oznacza po prostu, że jeśli klasa potomna musi zmienić lub rozszerzyć działanie jednej z dziedziczonych metod, to po prostu ją ponownie definiuje.

BRONEK: A w jaki sposób „każesz” obiektowi Ameba coś zrobić? Czy nie musisz wywołać jakiejś funkcji, o przepraszam — *metody* — i wskazać jej, którą figurę obrócić?

JUREK: I to właśnie jest najfajniejsza rzecz w programowaniu obiektowym. Kiedy trzeba obrócić, na przykład, trójkąt, kod programu wywołuje metodę obroc() obiektu trójkąta. Pozostała część programu tak naprawdę wcale nie interesuje się tym, w jaki sposób obraca się trójkąt. A kiedy trzeba dodać coś nowego do programu, tworzy się nową klasę dla nowego typu obiektu; dzięki temu ten **nowy obiekt będzie się zachowywać w sposób unikalny**.



Ta niepewność mnie zabije! Kto wygra Superfotel?



Ania z drugiego piętra.

(bez wiedzy kogokolwiek innego
Szef Projektu przekazał specyfikację
programu *trzem* programistom)

Co Ci się podoba w programowaniu obiektowym?

„Pomaga mi projektować programy w bardziej naturalny sposób. Obiekty wchodzące w skład programów mogą ewoluować.”

— Józef, 27, projektant oprogramowania

„To, że w razie konieczności dodania nowych możliwości nie muszę ingerować w kod, który już został napisany i przetestowany.”

— Bartek, 32, programista

„Podoba mi się, że dane i metody, które na tych danych operują, są zgrupowane w jednej klasie.”

— Jakub, 22, miłośnik piwa

„Podoba mi się możliwość wykorzystywania kodu w innych aplikacjach. Tworząc nową klasę, mogę ją napisać w sposób na tyle elastyczny, że będzie ją można wykorzystać w przyszłości w innych programach.”

— Krzysiek, 39, menedżer projektu

„Nie mogę uwierzyć, że Krzysiek coś takiego powiedział. Nie napisał nawet jednej linijki kodu od pięciu lat.”

— Dariusz, 44, pracuje dla Krzyska

„Oprócz możliwości zdobycia Superfotela?”

— Ania, 34, programistka

WYSIL SZARE KOMÓRKI

Nadszedł czas, żeby trochę rozruszać neurony.

Przeczytałeś właśnie opowieść o programie proceduralnym konkurencyjnym z programistą obiektowym. Przy okazji mógłś się przyjrzeć krótkiej prezentacji kluczowych pojęć związanych z programowaniem obiektowym, takich jak klasy, metody oraz atrybuty. Dalsza część rozdziału zostanie poświęcona dokładniejszej prezentacji klas i obiektów (do zagadnień dziedziczenia i przesłanania metod powrócimy w kolejnych rozdziałach).

Bazując na informacjach zdobytych do tej pory (a może także na wiedzy zdobytej podczas korzystania z innego języka orientowanego obiektowo), poświęć chwilkę na przemyślenie i próbę podania odpowiedzi na poniższe pytania:

Jakie są podstawowe zagadnienia, które należy przemyśleć, projektując klasy w języku Java? Jaki pytania należy sobie przy tym zadać? Gdybyś miał stworzyć listę rzeczy, które należy zrobić podczas projektowania nowych klas, to co znalazłbyś się na tej liście?

Metapoznaniowa podpowiedź

Jeśli utknąłeś i nie możesz znaleźć rozwiązania ćwiczenia, spróbuj porozmawiać o nim z sobą na głos. Mówienie (i słuchanie) aktywizuje różne części mózgu. Choć metoda ta daje najlepsze rezultaty, kiedy można porozmawiać z inną osobą, to jednak można także porozmawiać z ulubionym zwierzakiem. To właśnie w ten sposób nasz pies dowiedział się, co to jest polimorfizm.

Myślenie o obiektach

Projektując klasę, myśl o obiektach, które będą tworzone na podstawie tego typu. Pomyśl o:

- informacjach, jakie obiekt **zna**,
- czynnościami, jakie obiekt **wykonuje**.

Koszyk
zawartosc
dodajDoKoszyka() usunZKoszyka() sprawdz()

wie
wykonuje

Przycisk
etykieta
kolor
okreslKolor()
okreslEtykieta()
zwolnij()
wcisnij()

wie
wykonuje

Alarm
alarmCzas
alarmTryb
okreslCzasAlarmu()
pobierzCzasAlarmu()
czyAlarmUstawiony()
wstrzymaj()

wie
wykonuje

Informacje, jakie obiekt ma o sobie, są nazywane

- składowymi

Czynności, jakie obiekt jest w stanie wykonywać, są nazywane

- metodami

Informacje, jakie obiekt **wie**, na swój temat, nazywamy **składowymi**. Reprezentują one stan obiektu (dane) i mogą przyjmować unikalne wartości w każdym z obiektów danego typu.

Pamiętaj, że mówiąc o **kopii**, mamy na myśli **obiekt**.

Czynności, jakie obiekt **może wykonywać**, nazywamy **metodami**. Projektując klasę, należy określić, jakie informacje na swój temat obiekt musi znać, jak również zaprojektować metody, które będą operować na tych danych. Bardzo często się zdarza, że obiekty mają metody służące do ustawiania oraz odczytywania wartości składowych. Na przykład obiekt Alarm ma składową określającą czas alarmu oraz dwie metody służące do określania i pobierania tego czasu.

A zatem obiekty mają kopie, składowe oraz metody, jednak zarówno składowe, jak i metody stanowią część klasy.

Zmienne składowe (stan)
tytuł wykonawca

wie
wykonuje



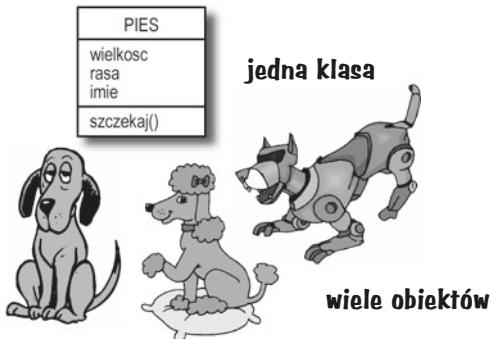
Poniżej wpisz, co obiekt Telewizor powinien wiedzieć i robić.



Telewizor

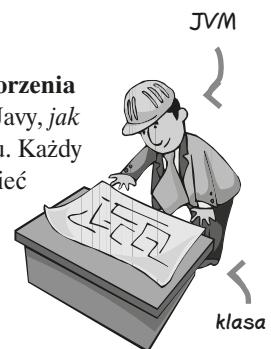
zmienne składowe
metody

Jaka jest różnica pomiędzy klasą a obiektem?



Klasa nie jest obiektem (jednak służy do ich tworzenia)

Klasa jest jak gdyby *matrycą* służącą do tworzenia obiektów. Informuje ona wirtualną maszynę Javy, *jak* należy utworzyć obiekt tego konkretnego typu. Każdy obiekt utworzony na podstawie klasy może mieć unikalne wartości składowych. Na przykład, można użyć klasy Przycisk do stworzenia kilkunastu różnych przycisków, z których każdy będzie mieć inny kolor, wielkość, kształt, etykietę i tak dalej.



**Wyobraź to sobie
w następujący sposób...**



Obiekt można porównać z jednym wpisem w książce adresowej.

Jedną z możliwych analogii obiektów są puste karteczki, które można wpisać do kłosego notatnika z adresami. Każda z takich karteczek ma takie same pola do wypełnienia (odpowiadające składowym obiektu). Wypełniając ją, tworzymy kopię (obiekt), a podane na niej informacje określają stan kopii.

Metody klasy to czynności, jakie można wykonywać na konkretnej kartecze; klasa NotatnikAdresowy mógłaby mieć następujące metody: pobierzNazwisko(), zmienNazwisko(), okreslNazwisko().

A zatem, każda karteczka może wykonywać te same operacje (pobrać zapisane na niej nazwisko, zmienić je i tak dalej), jednak każda z nich zawiera unikalne *informacje*, dostępne wyłącznie na niej.

Tworzenie pierwszego obiektu

A zatem, czego będziesz potrzebował do stworzenia i wykorzystania swojego pierwszego obiektu? Będą Ci potrzebne *dwie* klasy. Pierwsza będzie klasą obiektu, którego chcesz użyć (Pies, Budzik, Telewizor i tak dalej), natomiast druga posłuży do *przetestowania* nowej klasy. To właśnie w tej klasie *testującej* zostanie umieszczona metoda `main()`, a w niej będzie tworzony obiekt Twojej nowej klasy. Klasa testująca ma tylko jedno zadanie — *przetestować* metody i składowe obiektu Twojej nowej klasy.

Zaczynając od tego miejsca, w wielu przykładach przedstawionych w dalszej części książki będziesz mógł znaleźć dwie klasy. Pierwsza z nich będzie tą *właściwą* — czyli klasą, której obiektów chcemy używać; z kolei druga będzie klasą *testującą*, a jej nazwa będzie odpowiadać nazwie klasy właściwej z dodanym na końcu słowem **Tester**. Na przykład, jeśli stworzymy klasę **Bungee**, to będzie nam potrzebna także klasa **BungeeTester**. Wyłącznie klasa *<jakaś Tam Klasa>Tester* będzie posiadać metodę `main()`, a jedynym celem jej istnienia będzie stworzenie obiektów nowej klasy (nie klasy testującej) i wykorzystanie operatora kropki `(.)` w celu uzyskania dostępu do metod i składowych tych obiektów. Wszystkie te zasady staną się całkowicie jasne, gdy przeanalizujesz poniższe przykłady.

1 Napisz kod klasy.

```
class Pies {
    int wielkosc;
    String rasa;
    String imie;

    void szczekaj() {
        System.out.println("Chau! Chuuu!");
    }
}
```

składowe

metoda

klasa posiada tylko metodę `main()`
(w następnym kroku umieścimy w tej
metodzie jakiś kod)

2 Napisz klasę testującą (Tester).

```
class PiesTester {
    public static void main (String[] args) {
        // Tutaj umieścimy kod
        // testujący klasę Pies
    }
}
```

3 W klasie testującej stwórz obiekt i użyj jego składowych i metod.

```
class PiesTester {
    public static void main (String[] args) {
        Pies p = new Pies();
        p.wielkosc = 40;
        p.szzekaj();
    }
}
```

utwórz obiekt Pies

operator kropki

użyj operatora kropki `(.)` w celu określenia wielkości Psa

i wywołania jego metody `szzekaj()`

Jeśli już masz jakieś rozeznanie w programowaniu obiektowym, to będziesz wiedział, że nie używamy hermetyzacji. Tym zagadniением zajmiemy się w rozdziale 4.

Operator kropki `(.)`

Operator kropki `(.)` zapewnia dostęp do stanu oraz do zachowania obiektu (składowych i metod).

// tworzymy nowy obiekt

Pies p = new Pies();

// każemy psu szczać,
// dodając do zmiennej p
// operator kropki w celu
// wywołania metody
// szczekaj()

p.szzekaj();

// określamy wielkość psa,
// także tym razem
// używając operatora
// kropki `(.)`

p.wielkosc = 40;

Tworzenie i testowanie obiektów Film



```

class Film {
    String tytul;
    String rodzaj;
    int ocena;

    void odtworz() {
        System.out.println("Odtwarzamy film.");
    }
}

public class FilmTester {
    public static void main(String[] args) {
        Film pierwszy = new Film();
        pierwszy.tytul = "Przeminęło z hossą";
        pierwszy.rodzaj = "Tragedia";
        pierwszy.ocena = -2;
        Film drugi = new Film();
        drugi.tytul = "Matrix dla zuchwałych";
        drugi.rodzaj = "Komedia";
        drugi.ocena = 5;
        drugi.odtworz();
        Film trzeci = new Film();
        trzeci.tytul = "Byte Club";
        trzeci.rodzaj = "Tragedia, ale o wydźwięku optymistycznym";
        trzeci.ocena = 127;
    }
}

```



Zaostrz ołówek

FILM
tytuł
rodzaj
ocena
odtworz()

obiekt 1

tytuł
rodzaj
ocena

obiekt 2

tytuł
rodzaj
ocena

obiekt 3

tytuł
rodzaj
ocena

Klasa FilmTester tworzy trzy obiekty (kopie) klasy Film, a następnie, przy wykorzystaniu operatora kropki (.), przypisuje konkretne wartości ich składowym. Klasa ta wywołuje także metodę jednego z tych obiektów. Na rysunku z prawej strony w pustych miejscach wpisz wartości, jakie będą miały odpowiednie składowe obiektów pod koniec działania metody main().

Szybko! Opuszczamy metodę main!

Dopóki działasz w obrębie metody `main()`, dopóty tak naprawdę nie dotarłeś do Obiektowa. Wykonywanie operacji w tej metodzie jest dobre dla prostego programu testowego, jednak prawdziwe aplikacje pisane obiektowo wymagają, aby obiekty komunikowały się z innymi obiektami, a nie by były tworzone i testowane w jakiejś statycznej metodzie.

Dwa zastosowania metody main:

- do **testowania klas wykorzystywanych w aplikacji**,
- do **uruchamiania bądź wykonywania aplikacji**.

Prawdziwa aplikacja napisana w Javie to w zasadzie nic innego jak grupa obiektów, które komunikują się pomiędzy sobą. W tym przypadku *komunikowanie się* oznacza wywoływanie metod obiektów. Na poprzedniej stronie (jak również w 4. rozdziale książki) metoda `main()` została umieszczona w niezależnej klasie `Tester` i służyła do utworzenia i sprawdzenia metod i zmiennych innych klas. W rozdziale 6. przyjrzymy się wykorzystaniu klasy, w której metoda `main()` służy do uruchomienia prawdziwej aplikacji napisanej w Javie (czyli, między innymi, do utworzenia obiektów i zapewnienia im możliwości interakcji z innymi obiektami).

Jak na razie przedstawimy jednak prosty przykład tego, jak może działać prawdziwa aplikacja Javy. Ponieważ wciąż znajdujemy się na samym początku nauki Javy, nasz warsztat jest bardzo ograniczony, dlatego też uznasz zapewne, że przedstawiony program jest nieco głupkowy i nieefektywny. Możesz się zastanowić, co zrobić, aby go poprawić; swoją drogą, dokładnie to zrobimy w następnych rozdziałach. Nie przejmuj się, jeśli będziesz mieć trudności ze zrozumieniem fragmentów kodu — jego podstawowym celem jest przedstawienie komunikacji pomiędzy obiektami.

Zgadywanka

Podsumowanie:

Nasza gra — zgadywanka — wykorzystuje obiekt „gry” oraz trzy obiekty „graczy”. Gra generuje liczbę losową z zakresu od 0 do 9, a trzej gracze starają się ją odgadnąć. (Nikt nie mówił, że to ma być pasjonująca gra).

Klasy:

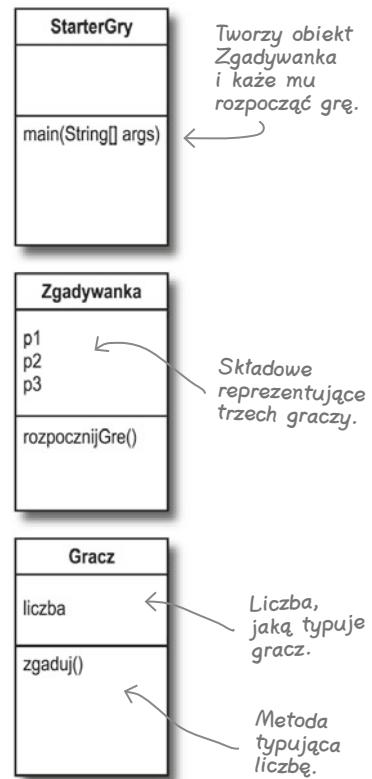
`Zgadywanka.class` `Gracz.class` `StarterGry.class`

Logika działania:

Działanie aplikacji rozpoczyna się w klasie `StarterGry`; klasa ta posiada metodę `main()`.

W metodzie `main()` jest tworzony obiekt `Zgadywanka`, a następnie zostaje wywołana metoda `rozpocznijGre()`.

Cała gra odbywa się wewnętrz metody `rozpocznijGre()` obiektu `Zgadywanka`. Metoda ta tworzy trzech graczy, po czym „wymyśla” losową liczbę (która gracze mają odgadnąć). Następnie metoda prosi graczy o odgadnięcie liczby, sprawdza podane przez nich wartości i wyświetla informacje o zwycięzcy (zwycięzach) albo prosi o ponowną próbę odgadnięcia.



```

class Zgadywanka {
    Gracz p1;
    Gracz p2;
    Gracz p3;

    public void rozpoczęjGre() {
        p1 = new Gracz();
        p2 = new Gracz();
        p3 = new Gracz();

        int typpl = 0;
        int typpp2 = 0;
        int typpp3 = 0;

        boolean p1odgadł = false;
        boolean p2odgadł = false;
        boolean p3odgadł = false;

        int liczbaOdgadywana = (int) (Math.random() * 10);
        System.out.println("Myśle o liczbie z zakresu od 0 do 9...");

        while(true) {
            System.out.println("Należy wytypować liczbę: " + liczbaOdgadywana);

            p1.zgaduj();
            p2.zgaduj();
            p3.zgaduj();

            typpl = p1.liczba;
            System.out.println("Gracz pierwszy wytypował liczbę: " + typpl);

            typpp2 = p2.liczba;
            System.out.println("Gracz drugi wytypował liczbę: " + typpp2);

            typpp3 = p3.liczba;
            System.out.println("Gracz trzeci wytypował liczbę: " + typpp3);

            if (typpl == liczbaOdgadywana) {
                p1odgadł = true;
            }
            if (typpp2 == liczbaOdgadywana) {
                p2odgadł = true;
            }
            if (typpp3 == liczbaOdgadywana) {
                p3odgadł = true;
            }

            if (p1odgadł || p2odgadł || p3odgadł) {
                System.out.println("Mamy zwycięzcę!");
                System.out.println("Czy gracz pierwszy wytypował poprawnie? " + p1odgadł);
                System.out.println("Czy gracz drugi wytypował poprawnie? " + p2odgadł);
                System.out.println("Czy gracz trzeci wytypował poprawnie? " + p3odgadł);
                System.out.println("Koniec gry.");
                break; // Gra skończona, zatem wychodzimy z pętli while
            } else {
                System.out.println("Gracze będą musieli spróbować jeszcze raz.");
            } // koniec if - else
        } // koniec while
    } // koniec metody rozpoczęjGre
} // koniec klasy

```

Klasa Zgadywanka ma trzy składowe służące do przechowywania trzech obiektów Gracz.

Utworzenie trzech obiektów Gracz i zapisanie ich w trzech składowych.

Deklaracja trzech zmiennych, w których będą przechowywane trzy liczby wytypowane przez poszczególnych graczy.

Deklaracja trzech zmiennych, które będą przechowywać wartości true (prawda) lub false (fałsz), w zależności od odpowiedzi konkretnego gracza.

Wyznaczenie liczby, jaką będą musieli odgadnąć gracze.

Wywołanie metody zgaduj() każdego z graczy.

Pobranie liczb wytypowanych przez każdego z graczy (wyników wywołania metody zgaduj()) poprzez odczytanie jej ze składowych obiektów graczy.

Sprawdzenie liczb wytypowanych przez graczy w celu określenia, czy odpowiadają one wyznaczonej liczbie. Jeśli gracz wytypował poprawnie, to odpowiedniej zmiennej przypisywana jest wartość true (pamiętaj, że domyślnie zmienna ta ma wartość false).

Jeśli gracz pierwszy LUB gracz drugi, LUB gracz trzeci odgadł... (operator || to logiczne LUB).

W przeciwnym przypadku pętla jest dalej realizowana, a gracze są proszeni o wytypowanie kolejnych liczb.

Uruchamianie zgadywanki

```
class Gracz {  
    int liczba = 0; // tu jest zapisywana typowana liczba  
  
    public void zgaduj() {  
        liczba = (int) (Math.random() * 10);  
        System.out.println("Typuję liczbę: " + liczba);  
    }  
}  
  
class StarterGry {  
    public static void main(String[] args) {  
        Zgadywanka gra = new Zgadywanka();  
        gra.rozpocznijGre();  
    }  
}
```



Java sama wynosi śmieci

Za każdym razem, gdy w Javie jest tworzony obiekt, trafia on do obszaru pamięci nazywanego **stertą**.

Wszystkie obiekty, niezależnie od tego, kiedy, jak i gdzie zostaną utworzone, zawsze są przechowywane na stercie. Jednak nie jest to sterta starej, zapomnianej pamięci; w Javie sterta jest także nazywana **stertą automatycznie odśmiecana**. Kiedy tworzyisz obiekt, Java rezerwuje na stercie obszar o wielkości odpowiadającej potrzebom konkretnego obiektu. Obiekt posiadający, dajmy na to, 15 składowych, będzie prawdopodobnie potrzebował więcej miejsca niż obiekt mający jedynie 2 składowe. Co się jednak dzieje, kiedy będzie trzeba odzyskać miejsce przydzielone na stercie? W jaki sposób można usunąć z niej obiekt, kiedy skończymy go już używać? To Java zarządza pamięcią za nas! Kiedy wirtualna maszyna Javy (JVM) „zauważ”, że obiekt nie będzie już mógł być wykorzystywany w programie, zostaje on uznany za *nadający się do odśmiecenia*, a kiedy w systemie zacznie brakować pamięci, zostanie uruchomiony odśmiecacz, który usunie z niej wszystkie nieosiągalne obiekty. W ten sposób pamięć zostanie zwolniona i będzie można ją ponownie wykorzystać. Więcej informacji na ten temat znajdziesz w kolejnym rozdziale.

Wyniki (za każdym razem będą inne)

```
C:\ Wiersz polecenia  
T:\HF-Java\Kody\R02>java StarterGry  
Myślę o liczbie z zakresu od 0 do 9...  
Należy wytypować liczbę: 6  
Typuję liczbę: 7  
Typuję liczbę: 3  
Typuję liczbę: 7  
Gracz pierwszy wytypował liczbę: 7  
Gracz drugi wytypował liczbę: 3  
Gracz trzeci wytypował liczbę: 7  
Gracze będą musieli spróbować jeszcze raz.  
Należy wytypować liczbę: 6  
Typuję liczbę: 4  
Typuję liczbę: 7  
Typuję liczbę: 4  
Gracz pierwszy wytypował liczbę: 4  
Gracz drugi wytypował liczbę: 7  
Gracz trzeci wytypował liczbę: 4  
Gracze będą musieli spróbować jeszcze raz.  
Należy wytypować liczbę: 6  
Typuję liczbę: 2  
Typuję liczbę: 3  
Typuję liczbę: 5  
Gracz pierwszy wytypował liczbę: 2  
Gracz drugi wytypował liczbę: 3  
Gracz trzeci wytypował liczbę: 5  
Gracze będą musieli spróbować jeszcze raz.  
Należy wytypować liczbę: 6  
Typuję liczbę: 7  
Typuję liczbę: 3  
Typuję liczbę: 0  
Gracz pierwszy wytypował liczbę: 7  
Gracz drugi wytypował liczbę: 3  
Gracz trzeci wytypował liczbę: 0  
Gracze będą musieli spróbować jeszcze raz.  
Należy wytypować liczbę: 6  
Typuję liczbę: 6  
Typuję liczbę: 6  
Typuję liczbę: 5  
Gracz pierwszy wytypował liczbę: 6  
Gracz drugi wytypował liczbę: 6  
Gracz trzeci wytypował liczbę: 5  
Mamy zwycięzcę!  
Czy gracz pierwszy wytypował poprawnie? true  
Czy gracz drugi wytypował poprawnie? true  
Czy gracz trzeci wytypował poprawnie? false  
Koniec gry.
```

Nie istnieja głupie pytania

P: Co zrobić, jeśli będę potrzebować globalnych zmiennych i metod? Jak to zrobić, jeśli wszystko musi być umieszczane wewnątrz klas?

O: W programach obiektowych pisanych w Javie nie istnieje pojęcie zmiennych lub metod „globalnych”. W szczególnych zastosowaniach istnieją jednak sytuacje, gdy chcemy, aby metoda (lub stała) była dostępna dla dowolnego fragmentu kodu działającego w dowolnej części programu. Przypomnij sobie metodę `random()` zastosowaną w programie krasomówczym, stanowi ona doskonały przykład metody, którą można wywołać w dowolnym miejscu programu. Albo, na przykład, stała `pi`. W rozdziale 10. dowieš się, że oznaczenie metod jako publiczne (przy użyciu słowa kluczowego `public`) i statyczne (przy użyciu słowa kluczowego `static`) sprawia, że zachowują się one jak metody „globalne” — będzie miał do nich dostęp dowolny kod działający w dowolnej klasie wchodzącej w skład programu. Z kolei, jeśli zmienna zostanie oznaczona jako publiczna, statyczna i finalna (odpowiednio przy użyciu słów kluczowych: `public`, `static` oraz `final`), to w efekcie stanie się ona globalnie dostępną stałą.

P: Ale co to za obiektowość, skoro wciąż można tworzyć zarówno funkcje, jak i dane globalne?

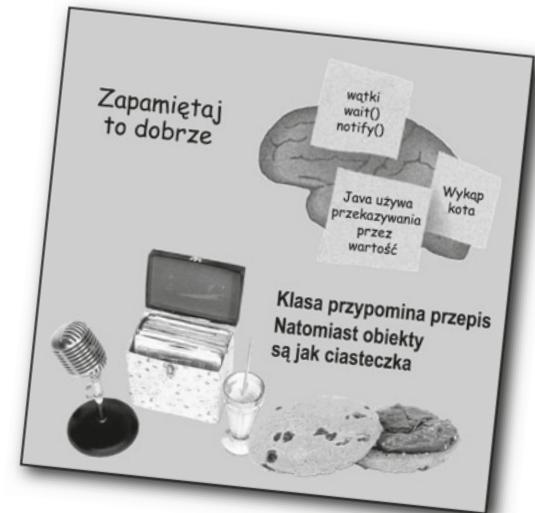
O: Przede wszystkim wszystko, co jest tworzone w Javie, musi być umieszczone w jakiejś klasie. Zatem stała `pi` oraz metoda `random()`, choć statyczne i publiczne, to jednak są zdefiniowane w klasie `Math`. Poza tym należy pamiętać, że takie dane i metody „globalne”, stanowią w Javie raczej wyjątek, a nie regułę. Stanowią one szczególny przypadek, w którym nie trzeba tworzyć wielu kopii obiektu, aby skorzystać z jego danych lub metod.

P: Czym jest program pisany w Javie? Co się w zasadzie rozpowszechnia?

O: Program napisany w Javie to grupa klas (a przynajmniej *jedna klasa*). W każdej aplikacji dokładnie jedna klasa musi mieć metodę `main()`, która służy do uruchamiania programu. A zatem Ty, jako programista, tworzysz jedną lub większą ilość klas. I właśnie one są rozpowszechniane jako program. Jeśli użytkownik końcowy nie posiada JVM, to do klas tworzących aplikację trzeba będzie dołączyć także środowisko wykonawcze Javy, dzięki któremu użytkownicy będą mogli uruchomić program. Dostępnych jest wiele programów instalacyjnych pozwalających na łączenie własnych klas z wieloma różnymi wersjami JVM (na przykład w zależności od docelowej platformy systemowej) i zapisywanie wszystkich niezbędnych plików na płytce CD-ROM. W ten sposób użytkownik końcowy może zainstalować odpowiednią wersję JVM (zakładając, że na jego komputerze wirtualna maszyna Javy nie jest jeszcze zainstalowana).

P: A co w sytuacji, gdy mają aplikację tworzy sto klas? Albo tysiąc? Czy dostarczanie tylu plików nie jest poważnym utrudnieniem? Czy nie można z nich zrobić jednego dużego, wykonywalnego pliku aplikacji?

O: Owszem, dostarczanie użytkownikowi końcowemu tak dużej ilości plików byłoby kłopotliwe. Na szczęście nie jest to konieczne. Można umieścić wszystkie pliki tworzące aplikację w jednym „archiwum Javy” — pliku `.jar` — bazującym na formacie archiwów `pkzip`. W pliku `jar` można umieścić odpowiednio sformatowany plik tekstowy stanowiący tak zwany *manifest* i określający, która klasa umieszczona w danym archiwum zawiera metodę `main()`, którą należy wywołać.



CELNE SPOSTRZEŻENIA

- Programowanie obiektowe pozwala na rozszerzanie programów bez konieczności modyfikowania przetestowanego wcześniej działającego kodu.
- W Javie cały tworzony kod jest umieszczany wewnątrz **klas**.
- Klasa opisuje, jak należy tworzyć obiekty danego typu. **Można ją zatem porównać do wzorca**.
- Obiekt potrafi o siebie zadbać; nie musisz ani wiedzieć, ani zaprzatać sobie głowy tym, jak obiekt coś robi.
- Obiekt **posiada** informacje i **wykonuje** czynności.
- Informacje, jakie obiekt ma na swój temat, są przechowywane w tak zwanych **składowych**. Reprezentują one *stan* danego obiektu.
- Czynności, jakie obiekt wykonuje, są nazywane **metodami**. Określają one *działanie* (lub *zachowanie*) obiektu.
- Tworząc klasę, można także stworzyć niezależną klasę testową służącą do tworzenia i sprawdzania obiektów nowej klasy.
- Klasa może **dzieziczyć** składowe i metody po bardziej abstrakcyjnych klasach **bazowych**.
- W czasie wykonywania program Javy jest w zasadzie grupą wzajemnie komunikujących się obiektów.



BĄDŹ kompilatorem



Każdy z plików przedstawionych na tej stronie stanowi niezależny kompletny plik źródłowy. Twoim zadaniem jest stać się kompilatorem i określić, czy przedstawione programy skompilują się czy nie. Jeśli nie można ich skompilować, to jak je poprawić? Jeśli można je skompilować, to jakie wygenerują wyniki?

A

```
class Magnetofon {  
    boolean mozeNagrywac = false;  
  
    void odtworzTasme() {  
        System.out.println("odtwarzam taśmę");  
    }  
  
    void nagrajTasme() {  
        System.out.println("nagrywam taśmę");  
    }  
}  
  
class MagnetofonTester {  
    public static void main(String[] args) {  
  
        m.mozeNagrywac = true;  
        m.odtworzTasme();  
  
        if (m.mozeNagrywac == true) {  
            m.nagrajTasme();  
        }  
    }  
}
```

B

```
class OdtwarzaczDVD {  
    boolean mozeNagrywac = false;  
  
    void nagrajPlyteDVD() {  
        System.out.println("nagrywam płytę DVD");  
    }  
}  
  
class OdtwarzaczDVDTester {  
    public static void main(String[] args) {  
  
        OdtwarzaczDVD o = new OdtwarzaczDVD();  
        o.mozeNagrywac = true;  
        o.odtwarzPlyteDVD();  
  
        if (o.mozeNagrywac == true) {  
            o.nagrajPlyteDVD();  
        }  
    }  
}
```



Ćwiczenie



Magnesiki z kodem

Działający program Java został podzielony na fragmenty, zapisany na małych magnesach, które przyczepiono do lodówki. Czy jesteś w stanie złożyć go z powrotem w jedną całość, tak aby wygenerował przedstawione poniżej wyniki? Niektóre nawiasy klamrowe spadły na podłogę i były zbyt małe, aby można je było podnieść; dlatego w razie potrzeby możesz je dodawać!

```
p.zagrajNaBebnie();
```

```
Perkusja p = new Perkusja();
```

```
boolean talerze = true;  
boolean beben = true;
```

```
void zagrajNaBebnie() {  
    System.out.println("bam, bam, baaaa-am-am");  
}
```

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

```
    if (p.beben == true) {  
        p.zagrajNaBebnie();  
    }
```

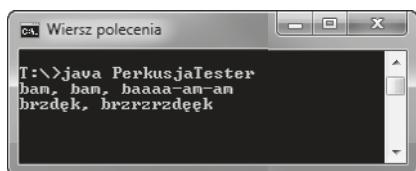
```
    p.beben = false;
```

```
class PerkusjaTester {
```

```
    p.zagrajNaTalerzach();
```

```
class Perkusja {
```

```
void zagrajNaTalerzach() {  
    System.out.println("brzdęk, brzrzrzdęęk");  
}
```



Zagadka. Zagadkowy basen



Zagadkowy basen



Twoim **zadaniem** jest wybranie fragmentów kodu z basenu i umieszczenie ich w miejscach kodu oznaczonych podkreśleniami. Każdy fragment kodu **może** być użyty więcej niż raz, a co więcej, nie wszystkie fragmenty zostaną wykorzystane. **Zadanie** polega na stworzeniu klasy, którą będzie można skompilować i która wygeneruje wyniki przedstawione poniżej. Nie daj się zwieść pozorom — ta zagadka jest trudniejsza, niż można by przypuszczać.

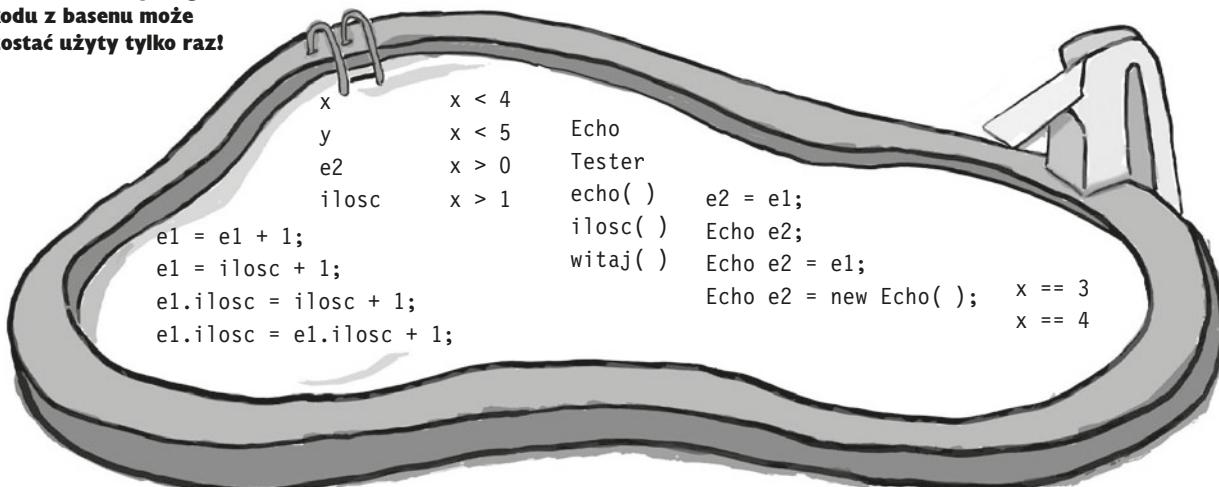
Wyniki:

```
Wiersz polecenia
I:\>java EchoTester
sieeeeeemasz...
sieeeeeemasz...
sieeeeeemasz...
sieeeeeemasz...
10
```

Pytanie dodatkowe!

Jak rozwiązałbyś zagadkę, gdyby w ostatnim wierszu wyników pojawiła się liczba **24**, a nie **10**?

Notatka: Każdy fragment kodu z basenu może zostać użyty tylko raz!



```
public class EchoTester {  
    public static void main(String[] args) {  
        Echo e1 = new Echo();  
  
        int x = 0;  
        while ( _____ ) {  
            e1.witaj();  
  
            if ( _____ ) {  
                e2.ilosc = e2.ilosc + 1;  
            }  
            if ( _____ ) {  
                e2.ilosc = e2.ilosc + e1.ilosc;  
            }  
            x = x + 1;  
        }  
        System.out.println(e2.ilosc);  
    }  
}
```

```
class _____ {  
    int _____ = 0;  
    void _____ {  
        System.out.println("sieeeeemasz... ");  
    }  
}
```



Kim jestem?



Grupa zamaskowanych komponentów Javy gra w grę towarzyską o nazwie „Zgadnij, kim jestem?”. Komponenty dają Ci podpowiedzi, a Ty na ich podstawie starasz się odgadnąć, kim one są. Załóż, że komponenty zawsze mówią prawdę. Jeśli mówią coś, co może być prawdą w odniesieniu do kilku z nich, wybierz wszystkie komponenty, dla których podane stwierdzenie jest prawdziwe. W pustych miejscach obok podanych podpowiedzi podaj nazwy komponentów biorących udział w zabawie. Odpowiedź na pierwszą podpowiedź podaliśmy sami.

W dzisiejszej zabawie udział biorą:

Klasa Metoda Obiekt Składowa

Jestem kompilowana na podstawie pliku .java.

klasa

Wartości moich składowych mogą się różnić od wartości składowych mojego brata bliźniaka.

Działam jak wzorzec.

Lubię działać.

Mogę mieć wiele metod.

Reprezentuję „stan”.

Mam swoje „działanie”.

Przebywam w obiektach.

Istnieję na stercie.

Służę do tworzenia kopii obiektów.

Mój stan może się zmieniać.

Deklaruję metody.

Mogę się zmieniać w trakcie działania programu.

Rozwiążania ćwiczeń



Ćwiczenie rozwiązanie

Magnesiki z kodem

```
class Perkusja {  
  
    boolean talerze = true;  
    boolean beben = true;  
  
    void zagrajNaBebnie() {  
        System.out.println("bam, bam, baaaa-am-am");  
    }  
  
    void zagrajNaTalerzach() {  
        System.out.println("brzdęk, brzrzdęk");  
    }  
  
}  
  
class PerkusjaTester {  
    public static void main(String[] args) {  
  
        Perkusja p = new Perkusja();  
        p.zagrajNaBebnie();  
        p.beben = false;  
        p.zagrajNaTalerzach();  
  
        if (p.beben == true) {  
            p.zagrajNaBebnie();  
        }  
    }  
}
```

```
Wiersz poleceń  
I:\>java PerkusjaTester  
bam, bam, baaaa-am-am  
brzdęk, brzrzdęk
```

Bądź kompilatorem

```
A class Magnetofon {  
    boolean mozeNagrywac = false;  
  
    void odtworzTasme() {  
        System.out.println("odtwarzam taśmę");  
    }  
  
    void nagrajTasme() {  
        System.out.println("nagrywam taśmę");  
    }  
}  
  
class MagnetofonTester {  
    public static void main(String[] args) {  
  
        Magnetofon m = new Magnetofon();  
        m.mozeNagrywac = true;  
        m.odtworzTasme();  
  
        if (m.mozeNagrywac == true) {  
            m.nagrajTasme();  
        }  
    }  
}
```

Mamy już wzorzec, teraz musimy stworzyć obiekt!

```
B class OdtwarzaczDVD {  
    boolean mozeNagrywac = false;  
    void nagrajPlyteDVD() {  
        System.out.println("nagrywam płytę DVD");  
    }  
    void odtworzPlyteDVD() {  
        System.out.println("odtwarzam płytę DVD");  
    }  
}  
  
class OdtwarzaczDVDTester {  
    public static void main(String[] args) {  
        OdtwarzaczDVD o = new OdtwarzaczDVD();  
        o.mozeNagrywac = true;  
        o.odtworzPlyteDVD();  
        if (o.mozeNagrywac == true) {  
            o.nagrajPlyteDVD();  
        }  
    }  
}
```

Wiersz kodu zawierający wywołanie o.odtworzPlyteDVD() nie skompiluje się, jeśli metoda nie będzie istnieć.



Rozwiążanie zagadki

Zagadkowy basen

```
public class EchoTester {
    public static void main(String[] args) {
        Echo e1 = new Echo();
        Echo e2 = new Echo(); // poprawna odpowiedź
        // -- lub --
        Echo e2 = e1; // odpowiedź na pytanie dodatkowe
        int x = 0;
        while ( x < 4 ) {
            e1.witaj();
            e1.ilosc = e1.ilosc + 1;
            if ( x == 3 ) {
                e2.ilosc = e2.ilosc + 1;
            }
            if ( x > 0 ) {
                e2.ilosc = e2.ilosc + e1.ilosc;
            }
            x = x + 1;
        }
        System.out.println(e2.ilosc);
    }
}
```

```
class Echo {
    int ilosc = 0;
    void witaj() {
        System.out.println("sieeeeeemasz... ");
    }
}
```

```
Wiersz poleceń
I:\>java EchoTester
sieeeeeemasz...
sieeeeeemasz...
sieeeeeemasz...
sieeeeeemasz...
10
```

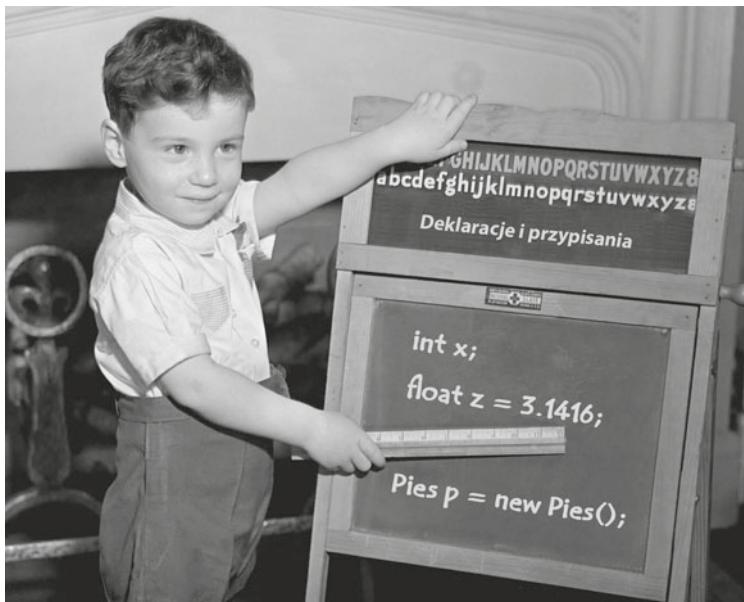
Kim jestem?

Jestem kompilowana na podstawie pliku .java.	<i>klasa</i>
Wartości moich składowych mogą się różnić od wartości składowych mojego brata bliźniaka.	<i>obiekt</i>
Działam jak wzorzec.	<i>klasa</i>
Lubię działać.	<i>obiekt, metoda</i>
Mogę mieć wiele metod.	<i>klasa, obiekt</i>
Reprezentuję „stan”.	<i>składowa</i>
Mam swoje „działanie”.	<i>obiekt, klasa</i>
Przebywam w obiektach.	<i>metoda, składowa</i>
Istnieję na stercie.	<i>obiekt</i>
Służę do tworzenia kopii obiektów.	<i>klasa</i>
Mój stan może się zmieniać.	<i>obiekt, składowa</i>
Deklaruję metody.	<i>klasa</i>
Mogę się zmieniać w trakcie działania programu.	<i>obiekt, składowa</i>

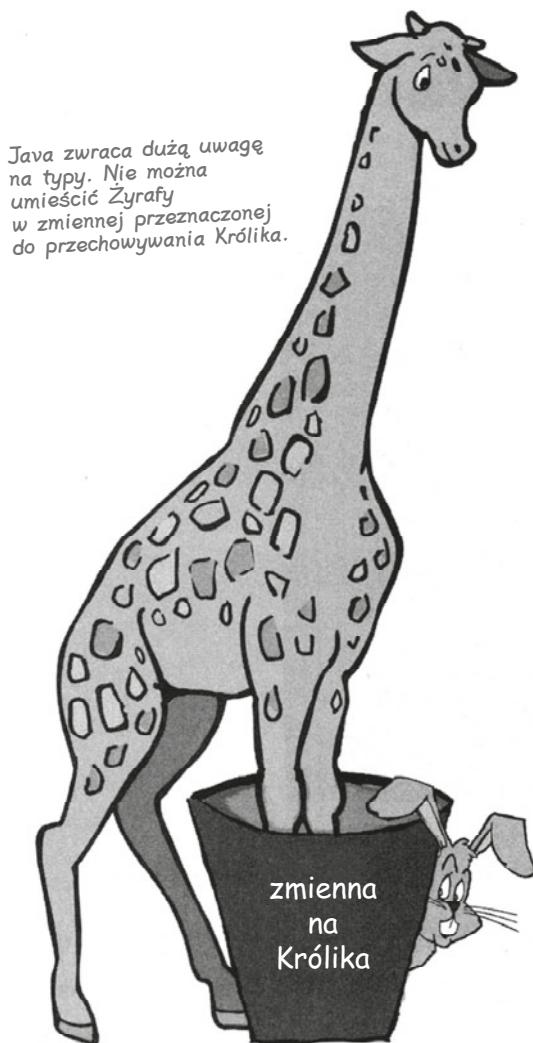
Notatka: Mówiąc, że zarówno klasy, jak i obiekty mają stan i działanie, mówimy o prawda definiowane w klasie, jednak mówimy, że także obiekty je „posiadają”. Jak na razie nie obchodzi nas, gdzie się one znajdują z technicznego punktu widzenia.

3. Typy podstawowe i odwołania

Poznaj swoje zmienne



Zmienne można podzielić na dwie kategorie: **zmienne typów podstawowych oraz odwołania**. Jak na razie zmiennych używałeś w dwóch przypadkach — do przechowywania **stanu** obiektu (składowe) oraz jako zmiennych **lokalnych** (czyli zmiennych deklarowanych wewnątrz *metod*). W dalszej części książki zmiennych będziemy używali także jako **argumentów** (czyli do przechowywania wartości przekazywanych z kodu wywołującego do metod) oraz jako **wartości wynikowych** (które są przekazywane z metody do kodu, który ją wywołał). Spotkałeś się już ze zmiennymi, w których deklaracjach pojawiały się **typy podstawowe**; przykładem takiego typu są zwyczajne liczby całkowite (typ `int`). Zetknąłeś się także ze zmiennymi, które były deklarowane jako dane bardziej **złożone**, na przykład jako tablice lub łańcuchy znaków (typ `String`). Ale przecież **musi być coś bardziej interesującego** niż liczby całkowite, łańcuchy znaków i tablice. Co należy zrobić, gdybyśmy chcieli stworzyć obiekt `WłaścicielZwierzaka` ze składową `Pies`? Albo obiekt `Samochód` ze składową `Silnik`? W tym rozdziale wyjaśnimy tajemnice typów danych w Javie i przyjrzymy się, co można zadeklarować jako zmienną, co w takiej zmiennej można **zapisać** i do czego jej można **użyć**. W końcu zobaczymy także, jak *naprawdę* wygląda życie na automatycznie odśmiecanej sterce.



Deklarowanie zmiennej

Java zwraca dużą uwagę na typy. Nie pozwoli na wykonanie jakiejś dziwnej i niebezpiecznej operacji, takiej jak umieszczenie odwołania do Żyrafy w zmiennej służącej do przechowywania Królików — co by się stało, gdyby takiemu „pseudokrólikowi” wydano polecenie `kicaj()`? Java nie pozwoli także na zapisanie wartości zmennoprzecinkowej w zmiennej całkowitej, chyba że *jawnie poinformujemy kompilator*, że wiemy o potencjalnej możliwości utraty precyzji liczby (czyli wszystkiego, co jest zapisane za przecinkiem dziesiętnym).

Kompilator może wykryć większość problemów:

```
Krolik marchewkozorca = new Zyrafa();
```

Nie oczekuj, że powyższy kod uda się skompilować. *I całe szczęście!*

Aby wszystkie te zabezpieczenia związane z typami danych mogły działać, trzeba zadeklarować wszystkie używane zmienne. Czy to jest liczba całkowita? Może Pies? A może pojedynczy znak? Zmienne można podzielić na dwie kategorie: zmienne *typów podstawowych* oraz *odwołania do obiektów*. Zmienne zaliczające się do tej pierwszej kategorii służą do przechowywania prostych wartości (które można uważać za ciągi bitów), takich jak liczby całkowite, wartości logiczne oraz liczby zmennoprzecinkowe. Z kolei odwołania, jak sama nazwa wskazuje, przechowują *odwołania do obiektów* (hmm... chyba niewiele Ci to wyjaśniło).

W pierwszej kolejności przyjrzymy się typom podstawowym, a następnie wyjaśnimy, czym tak naprawdę są odwołania. Niemniej jednak, niezależnie od typu, zawsze należy stosować dwie zasady związane z deklaracjami:

Zmienne muszą mieć typ

Oprócz typu każda zmienna musi mieć jakąś nazwę, którą będzie można używać w kodzie.

Zmienne muszą mieć nazwę

`int liczba;`
↑ ↓
typ nazwa

Notatka: Jeśli spotkasz się z wyrażeniem „obiekt **typu X**”, to powinieneś potraktować słowo *typ* jako synonim słowa *klasa*. (Wyjaśnimy to dokładniej w kolejnych rozdziałach).

„Proszę podwójną. Albo nie — całkowitą!”

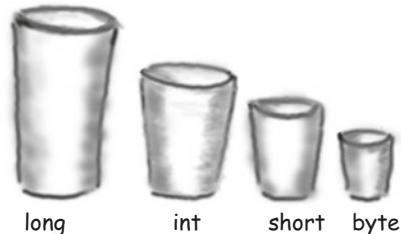
Myśląc o zmiennych w Javie, możesz je sobie wyobrażać jak kubeczki. Kubeczki z kawą, herbatą, ogromne kubki mieszczące wiele, wiele piwa, te gigantyczne teksturowe kubki z popcornem, które wszyscy kupują w kinach, kubeczki z falistymi, seksownymi uszczkami, kubki z metalowym rondem, których, jak powszechnie wiadomo, pod żadnym pozorem nie wolno wsadzać do kuchenki mikrofalowej.

Zmienna jest kubkiem. Pojemnikiem. Zmienna może coś zawierać.

Zmienne mają swój rozmiar i typ. W tym rozdziale w pierwszej kolejności przyjrzymy się zmiennym (kubeczkom) przechowującym dane **typów podstawowych**; następnie zajmiemy się kubeczkami przechowującymi *odwołania do obiektów*. Trzymajmy się na razie użytej powyżej analogii z kubeczkami — choć jest ona prosta, zapewni nam jednak wygodny sposób postrzegania zmiennych w przyszłości, gdy będziemy omawiać bardziej złożone zagadnienia. A dojdziemy do tego etapu już całkiem niedługo.

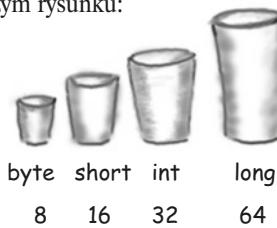
Typy podstawowe są jak kubki, które widujemy w kawiarniach. Jeśli kiedyś odwiedziłeś kawiarnię sieci Starbucks, to wiesz, o czym mówimy. Używane w nich kubki mają różne rozmiary, a każdy ma swoją nazwę, taką jak „krótki” czy też „długi”; można zatem poprosić o „wielki” kubek z połową kawy i połową bitej śmietany.

Wszystkie rodzaje kubków są ustawione na ladzie, dzięki czemu można wybrać odpowiedni:



Także w Javie podstawowe typy danych mają różne wielkości, a wielkości te są odpowiednio nazywane. Deklarując zmienną w Javie, należy określić jej typ. Cztery kubeczki przedstawione na rysunku obok reprezentują cztery podstawowe typy liczb całkowitych dostępne w Javie.

Każdy kubek zawiera wartość, a zatem odnosząc nasz przykład do zmiennych typów podstawowych, zamiast: „Kawę irlandzką w dużym kubku” można by powiedzieć „Poproszę zmienną całkowitą zawierającą wartość 90”. I wszystko by się zgadzało, gdyby nie jeden drobny szczegół... Otóż w Javie, oprócz typu, tworzone kubeczki muszą także mieć *nazwy*. A zatem, tak naprawdę trzeba by powiedzieć: „Poproszę zmienną całkowitą, o wartości 2489 i nazwie *wysokość*”. Każda dana typu podstawowego ma ściśle określona wielkość, czyli ilość bitów (która jest odpowiednikiem wielkości kubeczka). Wielkości sześciu liczbowych typów podstawowych stosowanych w Javie przedstawiliśmy na poniższym rysunku:



Typy podstawowe

Typ Ilość bitów Zakres wartości

Wartości logiczne i znaki

boolean	(zależy od JVM)	true lub false
char	16 bitów	0 do 65 535

Wartości liczbowe (wszystkie ze znakiem)

Liczby całkowite

byte	8 bitów	-128 do 127
short	16 bitów	-32 768 do 32 767
int	32 bity	-2 147 483 468 do 2 147 483 467
long	64 bity	– „bardzo dużo” do + „bardzo dużo”

Liczby zmiennoprzecinkowe

float	32 bity	różnie
double	64 bity	różnie

Deklaracje zmiennych typów podstawowych wraz z przypisaniem:

```
int x;
x = 234;
byte b= 89;
boolean jestCzadovo = true;
double d = 3243.98;
char c = 'f';
int z = x;
boolean hipHopRulez;
hipHopRulez = false;
boolean zasilanie;
zasilanie = jestCzadovo;
long duzy = 9257892;
float f = 32.4f;
```

Zwrócić uwagę na literę „f”. Musisz jej użyć, jeśli chcesz określić wartość typu zmiennoprzecinkowego o pojedynczej precyzyji (czyli float). Java traktuje wszystkie liczby z przecinkiem dziesiętnym jako wartości zmiennoprzecinkowe, o podwójnej precyzyji (typ double), chyba że na końcu wartości zostanie podana właściwie litera „f”.

Naprawdę nie chcesz niczego rozsypywać

Upewnij się, że wartość pasuje do typu zmiennej.



Nie można umieścić dużej wartości w małym kubeczkiku.

No... W zasadzie można, ale część wartości zostanie utracona. My nazywamy to *rozsypywaniem*. Kompilator stara się zapobiec takim sytuacjom, o ile tylko jest w stanie przewidzieć na podstawie kodu, że wartość nie zmieści się w używanym pojemniku (zmiennej albo, jak wolisz, kubeczku).

Na przykład nie można nasypać ilości herbaty odpowiadającej liczbie całkowitej do pojemnika dostosowanego do jednego bajta, co próbujemy zrobić w poniższym przykładzie:

```
int x = 24;  
byte b = x;  
// to nie przejdzie... !!
```

Możesz zapytać, dlaczego nie można tak zrobić? W końcu x ma wartość 24, a wartość ta, bez dwóch zdań, jest na tyle mała, że można ją zapisać w zmiennej typu byte. Owszem, Ty to wiesz, także *my* to wiemy, ale kompilator zwraca uwagę jedynie na fakt, że starasz się umieścić coś dużego w małym pojemniku i że *istnieje potencjalne* niebezpieczeństwo rozsypania. Nie oczekuj, że kompilator będzie wiedzieć, jaką jest wartość zmiennej *x*, nawet w przypadkach, gdy została jawnie podana w kodzie programu.

Zmiennej można przypisać wartość na kilka sposobów, w tym:

- podać *literał* za znakiem równości (*x = 12* czy *Dobrze = true* i tak dalej);
- zapisać w zmiennej wartość innej zmiennej (*x = y*);
- użyć wyrażenia łączącego oba powyższe rozwiązania (*x = y + 43*).

W poniższych wyrażeniach literaly zostały oznaczone pogrubioną kursywą:

```
int rozmiar = 32;      deklarujemy zmienną całkowitą typu int o nazwie rozmiar i przypisujemy jej wartość 32;  
char pierwsza = 'j';  deklarujemy zmienną znakową o nazwie pierwsza i zapisujemy w niej wartość 'j';  
double d = 3242.34;    deklarujemy zmienną zmiennoprzecinkową o podwójnej precyzyji o nazwie d i zapisujemy w niej wartość 3242.34;  
boolean czyToWariat;   deklarujemy zmienną logiczną o nazwie czyToWariat (i, jak na razie, nie przypisujemy jej żadnej wartości);  
czyToWariat = true;   przypisujemy wartość true zadeklarowanej wcześniej zmiennej czyToWariat;  
int y = x + 456;     deklarujemy zmienną całkowitą typu int o nazwie y i przypisujemy jej wartość będącą sumą wartości zmiennej x (jaka by ona nie była) i liczby 456.
```



Zaostrz ołówek

Kompilator nie pozwoli Ci zapisać wartości z większego kubeczka w mniejszym kubeczkiku. A na odwóz? Czy można przelać zawartość małego kubeczka do dużego kubka? **Oczywiście**.

Bazując na posiadanych informacjach o podstawowych typach danych oraz ich wielkościach, sprawdź, czy jesteś w stanie określić, które z poniższych instrukcji są poprawne, a które są błędne. Nie opisaliśmy jeszcze wszystkich zasad związanych z przypisywaniem, a zatem określając niektóre z poniższych przypadków, będziesz musiał zdać się na własny osąd.

Podpowiedź: Kompilator zawsze zgłasza błędy, jeśli istnieje jakieś zagrożenie.

Na poniższej liście **zakreś** wszystkie instrukcje, które byłyby poprawne, gdyby wszystkie instrukcje znajdowały się w jednej metodzie:

1. **int x = 34.5;**
2. **boolean boo = x;**
3. **int g = 17;**
4. **int y = g;**
5. **y = y + 10;**
6. **short s;**
7. **s = y;**
8. **byte b = 3;**
9. **byte v = b;**
10. **short n = 12;**
11. **v = n;**
12. **byte k = 128;**

Trzymaj się z daleka od tego słowa kluczowego!

Wiesz już, że każda zmienna musi mieć typ i nazwę.

Poznałeś też wszystkie podstawowe typy danych.

Ale co może być nazwą zmiennej? W tym przypadku reguły są proste. Nazwy klas, metod oraz zmiennych można określać, kierując się następującymi zasadami (prawdziwe zasady są nieco bardziej elastyczne, jednak te podane poniżej zapewnią Ci większe bezpieczeństwo):

- Nazwa musi zaczynać się od litery, znaku podkreślenia (_) lub znaku dolara (\$). Nie może natomiast zaczynać się od cyfry.
- Po pierwszym znaku, w nazwie można także umieszczać cyfry. Cyfra nie może jedynie znaleźć się na początku nazwy.
- Nazwa może mieć dowolną postać, pod warunkiem, że nie narusza dwóch wcześniejszych reguł i pod warunkiem, że nie jest jednym z zarezerwowanych słów Javy.

Słowa zarezerwowane to (między innymi) słowa kluczowe rozpoznawane przez kompilator Javy. A jeśli naprawdę zależy Ci, aby zagrać w grę „zmylenie kompilatora”, to spróbuj zastosować jedno z tych słów jako nazwę zmiennej.

Już wcześniej spotkałeś się z niektórymi zarezerwowanymi słowami języka, przy okazji tworzenia pierwszej klasy głównej:

```
public      static      void      ←
           ↙
boolean   char   byte   short   int   long   float   double   ↘
           ↙
```

Nazwy typów podstawowych także są słowami zarezerwowanymi:

```
boolean   char   byte   short   int   long   float   double   ↘
```

Istnieje jednak znacznie więcej słów zarezerwowanych, o których jeszcze nie wspominaliśmy. Nawet jeśli nie masz wiedzieć, co one znaczą, to jednak powinieneś wiedzieć, że nie możesz ich wykorzystywać do własnych celów. *Pod żadnym pozorem nie staraj się zapamiętać w tej chwili wszystkich podanych poniżej zarezerwowanych słów Javy.* Aby zrobić dla nich miejsce w swoim mózgu, musiałbyś prawdopodobnie zapomnieć o czymś innym. Na przykład o tym, gdzie zaparkowałesz samochód. Nie przejmuj się. Kiedy dotrzesz do końca książki, prawdopodobnie będziesz pamiętała wszystkie te słowa.

Tabela słów zarezerwowanych

boolean	byte	char	double	float	int	long	short	public	private
protected	abstract	final	native	static	strictfp	synchronized	transient	volatile	if
else	do	while	switch	case	default	for	break	continue	assert
class	extends	implements	import	instanceof	interface	new	package	super	this
catch	finally	try	throw	throws	return	void	const	goto	enum

Słowa kluczowe oraz inne zarezerowane słowa Javy. Nie używaj ich jako nazw, bo kompilator zrobi Ci piekielną awanturę.



Kontrolowanie obiektu Pies

Wiesz już, jak zadeklarować zmienną typu podstawowego i jak przypisać jej wartość. A co z innymi typami danych? Innymi słowy — *co z obiektemi?*

- Właściwie to nie ma czegoś takiego jak zmienna obiektowa.
- Są jedynie zmienne referencyjne (odwołania).
- Takie zmienne przechowują bity określające, w jaki sposób można dostać się do obiektu.
- W takich zmiennych nie są zapisywane same obiekty, lecz jedynie coś, co przypomina wskaźniki na obiekty. Albo adresy. Z tą różnicą, że w Javie tak naprawdę nie wiemy, co jest zapisywane w zmiennych referencyjnych. Wiemy tylko tyle, że niezależnie od tego, co to jest, to reprezentuje jeden i tylko jeden obiekt. Wiemy także, że JVM „wie”, jak wykorzystać to odwołanie w celu dotarcia do obiektu.

Nie można zapisać obiektu w zmiennej. Często myślimy w taki sposób... Mówimy coś w stylu: „Przekazałem String do metody System.out.print()”. Albo: „Metoda zwraca obiekt Pies”. Albo: „Zapisałem obiekt Kotek w zmiennej o nazwie kotekAli”.

Jednak takie określenia niedokładnie odpowiadają temu, co się w rzeczywistości dzieje. Nie ma gigantycznych, elastycznych kubków, które mogą się rozszerzać, dopasowując swoje wymiary do wielkości dowolnego obiektu. Obiekty istnieją tylko i wyłącznie w jednym miejscu — na automatycznie odśmiecanej sterce! (W dalszej części rozdziału znajdziesz więcej informacji na ten temat).

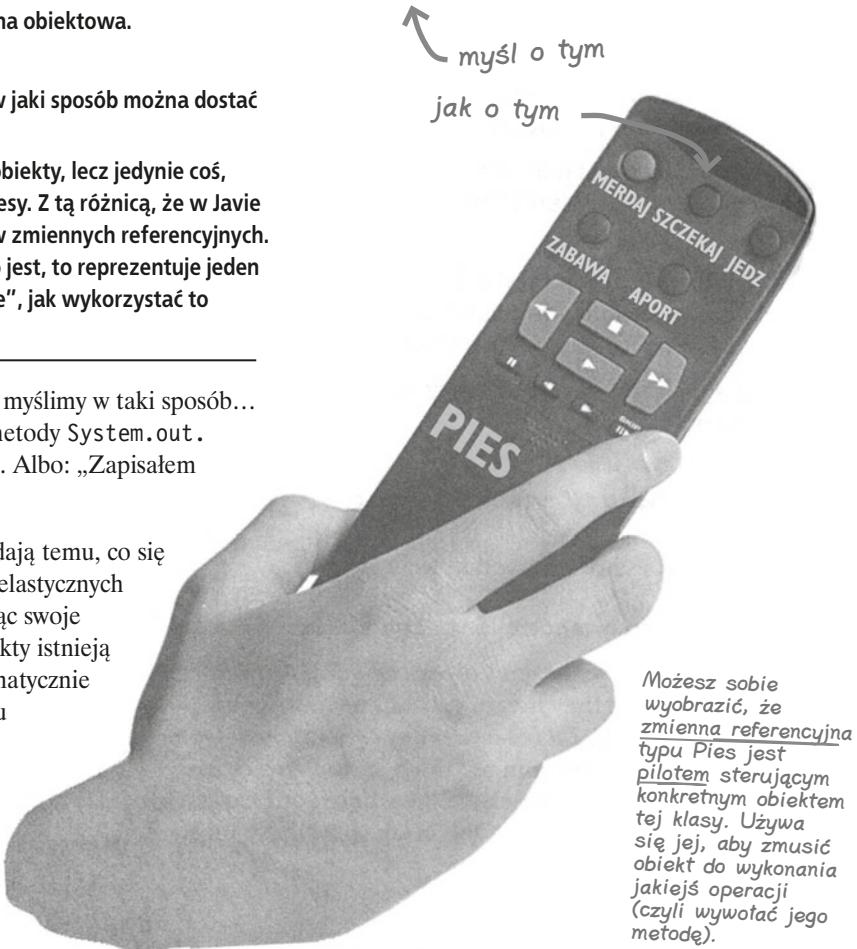
Zmienne typów podstawowych zawierają ciągi bitów reprezentujących faktyczną *wartość* zmiennej, natomiast zmienne referencyjne zawierają bity reprezentujące *sposób dotarcia do obiektu*.

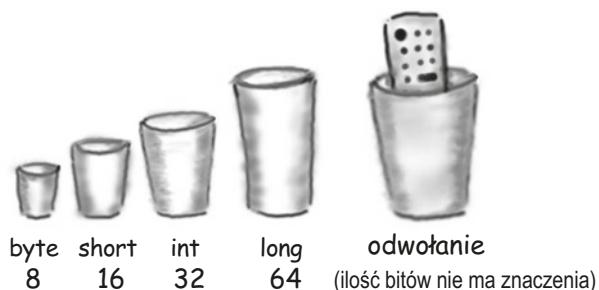
Można użyć operatora kropki (.) wraz ze zmienną referencyjną, aby powiedzieć coś w stylu: „użyj zmiennej *przed* kropką, aby dostać się do tego, co jest *po* kropce”. Na przykład:

mojPies.szczerkaj();

oznacza: „użyj obiektu, do którego odwołuję się zmienna mojPies, aby wywołać metodę szczerkaj()”. Użycie operatora kropki wraz ze zmienną referencyjną możesz sobie wyobrazić jako naciskanie przycisku na pilocie sterującym konkretnym obiektem.

**Pies p = new Pies();
p.szczerkaj();**





Odwołanie do obiektu to jedynie inna wartość zmiennej

To coś, co można umieścić w kubeczku, tylko że w tym przypadku wartość jest pilotem, który zdalnie steruje konkretnym obiektem.

Zmienna typu podstawowego

`byte x = 7;`

W zmiennej zapisywane są bity reprezentujące wartość 7 (00000111).



Odwołanie

`Pies mojPies = new Pies();`

W zmiennej zapisywane są bity reprezentujące sposób, w jaki można dotrzeć do obiektu Pies.

Natomiast sam obiekt Pies nie jest zapisywany w zmiennej!



W przypadku zmiennych typów podstawowych wartość zmiennej jest... po prostu wartością (5, -26.7, 'a').

W przypadku odwołań wartością zmiennej jest... ciąg bitów określający sposób dotarcia do konkretnego obiektu.

Nie wiesz (ani Cię nie obchodzi), jak konkretne wirtualne maszyny Javy implementują odwołania. Jasne, mogą to być wskaźniki na wskaźniki na... jednak nawet gdybyś wiedział, to i tak nie byłbyś w stanie wykorzystać tych bitów do niczego innego niż uzyskanie dostępu do obiektu.

Nie obchodzi nas, ile zer i jedynek wchodzi w skład odwołania. To zależy od wirtualnej maszyny Javy i fazy księżyca.

Trzy etapy deklarowania, tworzenia i przypisywania obiektu.

1 **2** **3**
`Pies mojPies = new Pies();`

- 1** Zadeklarowanie zmiennej referencyjnej.

`Pies mojPies = new Pies();`

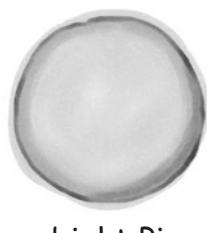
Powyższa instrukcja nakazuje JVM przydzielenie pamięci na zmienną referencyjną i nadaje tej zmiennej nazwę mojPies. Już do końca swojego istnienia zmienna ta będzie typu Pies. Innymi słowy pilot będzie miał przyciski do sterowania obiektami Pies, a nie Kot, Button czy Socket.



- 2** Utworzenie obiektu

`Pies mojPies = new Pies();`

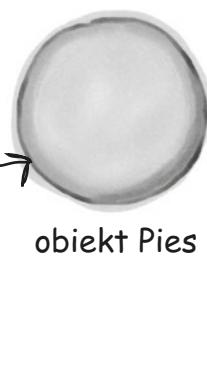
Ta instrukcja nakazuje JVM przydzielenie na stercie obszaru pamięci dla nowego obiektu Pies (w dalszej części książki, a zwłaszcza w rozdziale 9., dowiesz się znacznie więcej na temat tego procesu).



- 3** Utworzenie obiektu

`Pies mojPies = new Pies();`

Powyższa instrukcja zapisuje nowy obiekt Pies w zmiennej mojPies. Innymi słowy — *programuje pilota do obiektu*.



Odwołania do obiektów

Nie istnieja
głupie pytania

P: Jak duże jest odwołanie?

O: Nie wiadomo. Nie wiadomo, jak są reprezentowane odwołania, chyba że masz przyjaciela, który pracuje w grupie programistów zajmujących się tworzeniem Javy. Gdzieś tam są używane wskaźniki, ale nie ma do nich dostępu. Poza tym dostęp do tych wskaźników nie jest potrzebny. (No dobrze, jeśli nalegasz, to z powodzeniem mógłbyś sobie wyobrazić, że odwołania są wartościami 64-bitowymi). Jednak zastanawiając się nad zagadnieniami przydzielania pamięci, Naprawdę Ważnym Pytaniem jest to, ile obiektów (a nie odwołań) tworzysz oraz ile zajmują one miejsca („one” czyli obiekty).

P: Czy oznacza to, że wszystkie odwołania do obiektów mają tę samą wielkość niezależnie od wielkości obiektów, do których się odwołują?

O: Dokładnie. W danej wirtualnej maszynie Javy wszystkie odwołania będą miały tę samą wielkość, niezależnie od wielkości obiektów, do których się odwołują. Jednak każda wirtualna maszyna Javy może reprezentować odwołania w inny sposób, a zatem odwołania w jednej wirtualnej maszynie Javy mogą być mniejsze lub większe od odwołań w innych wirtualnych maszynach.

P: Czy na odwołaniach można wykonywać jakieś operacje arytmetyczne, na przykład je inkrementować? No wiesz... tak jak w C?

O: Nie. Jeszcze raz powtórz razem z nami — „Java to nie jest C”.



Java bez tajemnic

**W tym tygodniu przeprowadzimy wywiad z:
Odwołaniem do obiektu**

Autorzy: A zatem, powiedz nam, jak to jest być odwołaniem?

Odwołanie: W zasadzie, całkiem prosto. Jestem pilotem i można mnie zaprogramować, aby sterowało różnymi obiektami.

Autorzy: Czy masz na myśli sterowanie różnymi obiektami podczas działania? Czyli że teraz odwołujesz się do obiektu Pies, a za pięć minut możesz dowoływać się do obiektu Samochód?

Odwołanie: No jasne, że nie. Kiedy zostanę zadeklarowany to już koniec — po sprawie. Jeśli jestem pilotem do obiektu Pies, to nigdy nie będę mogły wskazywać (ups! — mój błąd, przepraszam, nie powiniem mówić „wskazywać”), miałem na myśli odwoływać się, do jakiegokolwiek innego obiektu.

Autorzy: Ale czy to oznacza, że możesz się odwoływać tylko do jednego obiektu Pies?

Odwołanie: Nie. Mogę odwoływać się do jednego obiektu Pies, a za pięć minut do zupełnie innego. O ile tylko jest to Pies. Można mnie przekierować na inny obiekt (to tak jakby przeprogramować pilota, aby sterował innym telewizorem). Chyba że... zresztą mniejsza z tym.

Autorzy: Nie, nie. Proszę, co chciałeś powiedzieć?

Odwołanie: Nie przypuszczam, żebyście aktualnie chcieli wchodzić w takie szczegóły, ale opiszę to w sposób skrócony. Otóż jeśli zostanę oznaczony jako finalne, to po przypisaniu mi jakiegoś obiektu Pies, nie będzie mnie można przeprogramować na żadnego innego Psa. Innymi słowy, nie będzie mi można przypisać żadnego innego obiektu.

Autorzy: Masz rację, nie chcemy teraz o tym rozmawiać. No dobrze, a zatem, jeśli nie jesteś finalne, to możesz odwoływać się do jednego obiektu Pies, a chwilę potem do innego. Ale czy naprawdę nie możesz odwoływać się do niczego innego? Czy to możliwe, abyś nie było zaprogramowanie do niczego?

Odwołanie: Cóż, to możliwe, ale rozmowa o tym jest krępująca.

Autorzy: Dlaczego?

Odwołanie: Gdyż to oznacza, że jestem puste — null — rozumiesz? A to mnie przygnębia.

Autorzy: Chodzi ci o to, że nie masz wartości?

Odwołanie: Ależ skąd — null to jest wartość. Wciąż jestem pilotem... ale wyobraź sobie, że kupiłeś nowy, uniwersalny pilot, ale w domu nie masz żadnego telewizora. W takiej sytuacji nie jestem zaprogramowane do obsługi czegokolwiek. Mogą naciskać moje przyciski cały dzień, a i tak nie stanie się nic ciekawego. W takich sytuacjach czuję się takie... niepotrzebne. Marnowanie bitów. Fakt, że niezbyt wielu bitów, ale zawsze... Lecz nie to jest najgorsze. Jeśli jestem ostatnim odwołaniem do konkretnego obiektu, to po zapisaniu we mnie null (czyli usunięciu ze mnie oprogramowania), już nikt nie będzie w stanie do niego dotrzeć.

Autorzy: A to faktycznie niedobrze, ponieważ...

Odwołanie: Musisz pytać? Ja się tu staram, tworzę z obiektem więzy, intymne połączenie, a potem ktoś je nagle, brutalnie zrywa. I nigdy więcej już tego obiektu nie zobaczę, gdyż zostaje on oznaczony jako nadający się do... (producent — tu należy odtworzyć dramatyczny podkład muzyczny) odśmiecenia. Chlip, chlip... Ale czy sądzisz, że to kiedykolwiek obchodziło programistów? Chlip, chlip... Dlaczego, dlaczego nie mogę być zmienną typu podstawowego? Nienawidzę bycia odwołaniem. Ta odpowiedzialność, te wszystkie przerwane powiązania...

Życie na odśmiecanej stercie

`Nowela b = new Nowela();`

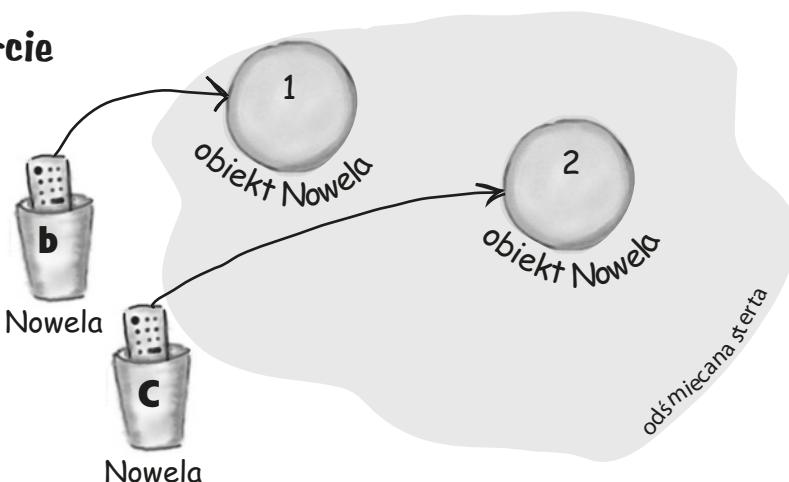
`Nowela c = new Nowela();`

Deklarujemy dwie zmienne, które mają przechowywać odwołania do obiektów Nowela. Następnie są tworzone dwa nowe obiekty Nowela, które zostają zapisane w zmiennych.

Oba utworzone obiekty Nowela istnieją na stercie.

Odwołań: 2

Obiektów: 2



`Nowela d = c;`

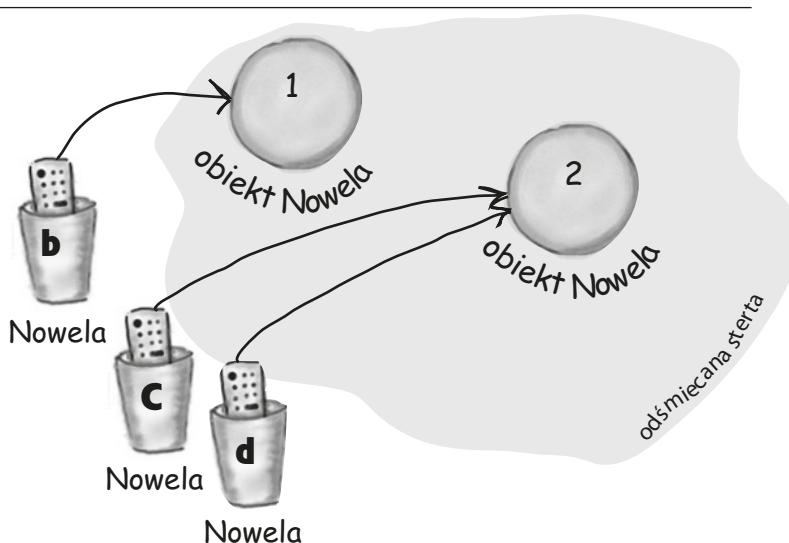
Deklarujemy nową zmienną, która ma przechowywać odwołanie do obiektu Nowela. Jednak zamiast tworzenia nowego, trzeciego już obiektu Nowela, w nowej zmiennej `d` zapisujemy wartość zmiennej `c`. Ale co to oznacza? Można by to porównać ze stwierdzeniem: „Pobierz bity stanowiące zawartość zmiennej `c`, skopiuj je, i tą kopię zapisz w zmiennej `d`”.

Teraz zarówno `c`, jak i `d` odwołują się do tego samego obiektu.

Zmienne `c` i `d` zawierają dwie różne kopie tej samej wartości – jak gdyby dwa piloty obsługujące ten sam odbiornik telewizyjny.

Odwołania: 3

Obiekty: 2



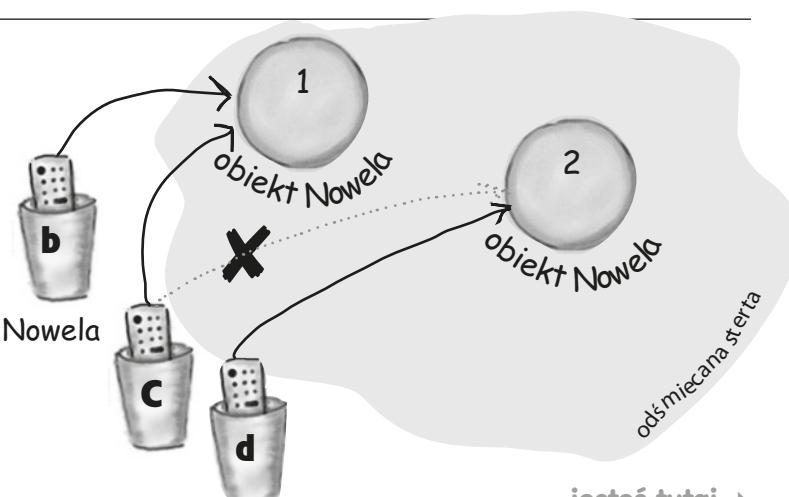
`c = b;`

Zapisujemy wartość zmiennej `b` w zmiennej `c`. Teraz już wiesz, co to oznacza. Bity stanowiące zawartość zmiennej `b` są kopowane, a ich kopia zostaje zapisana w zmiennej `c`.

Zarówno `b`, jak i `c` odwołują się do tego samego obiektu.

Odwołania: 3

Obiekty: 2



Obiekty na stercie

Życie na odśmiecanej stercie

Nowela b = new Nowela();

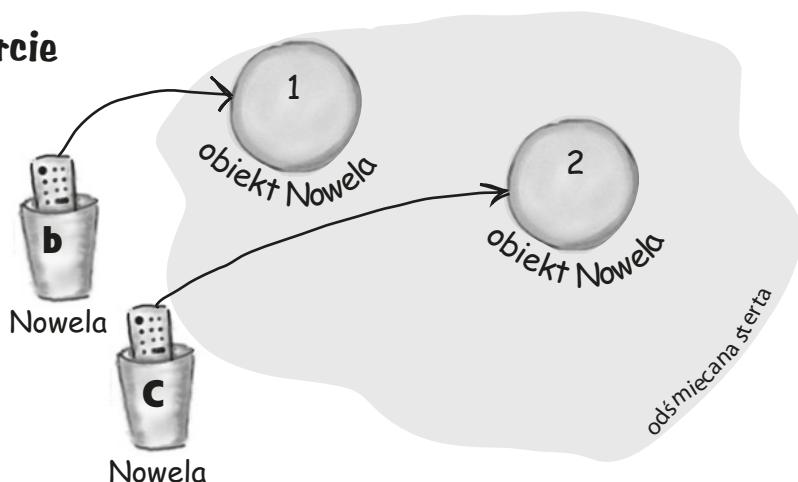
Nowela c = new Nowela();

Deklarujemy dwie zmienne, które będą przechowywać odwołania do obiektów Nowela. Następnie są tworzone dwa obiekty Nowela, które zostają zapisane w zmiennych.

W efekcie na stercie pojawiają się dwa obiekty Nowela.

Aktywne odwołania: 2

Osiągalne obiekty: 2



b = c;

Zapisujemy wartość zmiennej **c** w zmiennej **b**. Bity stanowiące zawartość zmiennej **c** są kopiowane i zapisywane w zmiennej **b**. Obie zmienne mają w efekcie takie same wartości.

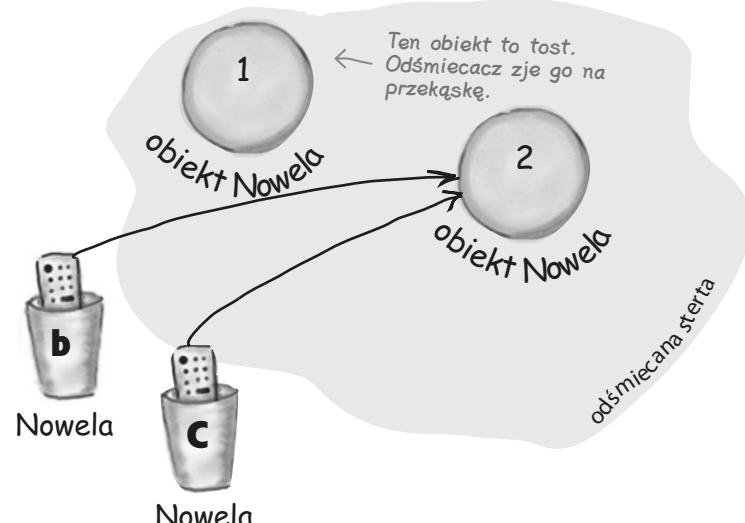
Zarówno zmienna b, jak i c odwołują się do tego samego obiektu. Obiekt oznaczony cyfrą 1 zostaje porzucony i uznany za nadający się do odśmieczenia.

Aktywne odwołania: 2

Osiągalne obiekty: 1

Porzucone obiekty: 1

Nie istnieją już żadne odwołania do obiektu, do którego początkowo odwoływała się zmieni **b**, czyli do obiektu oznaczonego cyfrą 1. A zatem obiekt ten jest *nieosiągalny*.



c = null;

Przypisujemy zmiennej **c** wartość null. W ten sposób **c** staje się pustym odwołaniem, co oznacza, że nie odwołuje się do żadnego obiektu. Jednak **c** wciąż jest zmienną, w której w dowolnej chwili można zapisać inne odwołanie do dowolnego obiektu Nowela.

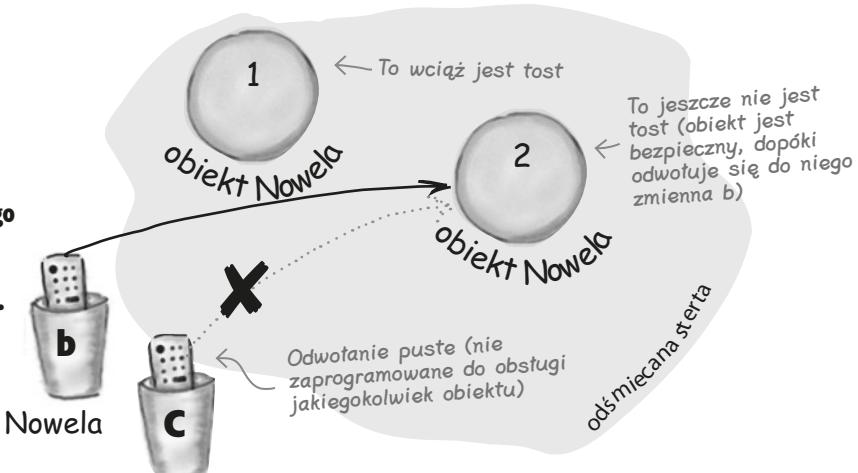
Wciąż istnieje odwołanie do obiektu oznaczonego na rysunku cyfrą 2 (jest ono przechowywane w zmiennej b) i póki istnieje, obiekt ten nie zostanie uznany za nadający się do odśmieczenia.

Aktywne odwołania: 1

Puste odwołania: 1

Osiągalne obiekty: 1

Porzucone obiekty: 1



Tablica jest jakby tacą z kubkami

- 1** Deklarujemy zmienną tablicową przystosowaną do przechowywania liczb całkowitych. Taka zmienna tablicowa zawiera odwołanie do obiektu tablicy.

```
int[] liczby;
```

- 3** Każdemu elementowi tablicy przypisujemy wartość całkowitą.

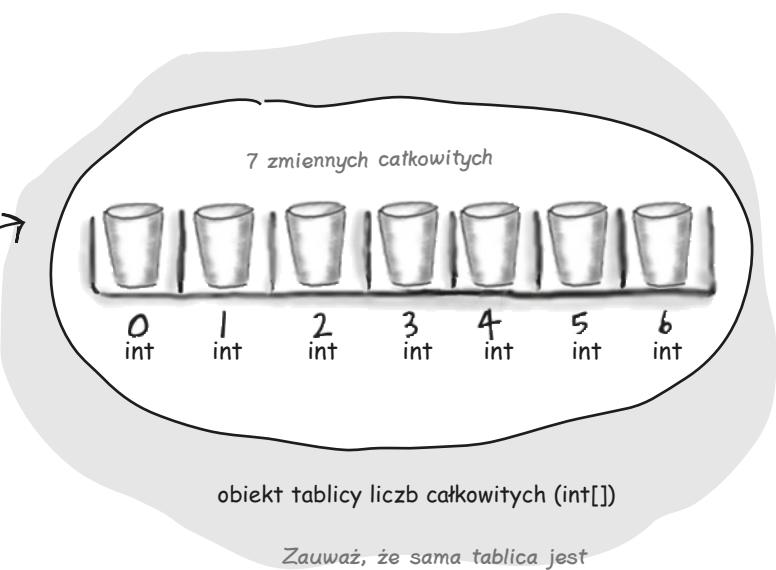
Pamiętaj, że elementy tablicy liczb całkowitych to zwyczajne zmienne, w których można zapisywać liczby całkowite.

```
7 zmiennych całkowitych
    {
        liczby[0] = 6;
        liczby[1] = 19;
        liczby[2] = 44;
        liczby[3] = 42;
        liczby[4] = 10;
        liczby[5] = 20;
        liczby[6] = 1;
    }
```



- 2** Tworzymy nową tablicę liczb całkowitych o długości 7 i zapisujemy ją w zadeklarowanej wcześniej (int []) zmiennej liczby.

```
liczby = new int[7];
```



Tablice także są obiektami

Standardowa biblioteka Java zawiera wiele wyrafinowanych struktur danych, takich jak mapy, drzewa oraz zbiory (informacje na ich temat można znaleźć w dodatku B), jednak tablice są niezastąpione w sytuacjach, gdy potrzebna nam jest szybka, uporządkowana i efektywna lista danych. Tablice zapewniają możliwość uzyskania szybkiego dostępu do dowolnego elementu tablicy, którego położenie określone jest za pomocą indeksu.

Każdy element tablicy jest po prostu zmienną. Innymi słowy może to być dowolna dana jednego z typów podstawowych lub odwołanie. Wszystko, co można by zapisać w zmiennej danego

typu, można też umieścić w elemencie tablicy tego samego typu. A zatem, w przypadku tablicy liczb całkowitych (int []) każdy jej element może zawierać liczbę całkowitą. Z kolei w przypadku tablicy obiektów Pies (Pies []) każdy jej element może zawierać... obiekt Pies? Nie pamiętasz, że zmienne referencyjne zawierają nie same obiekty, lecz odwołania do nich? A zatem w tablicy typu Pies każdy element takiej tablicy może zawierać pilot do obiektu Pies. Oczywiście wciąż musimy stworzyć obiekty Pies... temu zagadnieniu przyjrzymy się na następnej stronie. Koniecznie musisz zwrócić uwagę na pewien aspekt tablic przedstawiony

Zauważ, że sama tablica jest obiektem, choć jej elementy są zmiennymi typów podstawowych.

na powyższym rysunku — tablica jest obiektem, nawet jeśli jest to tablica typów podstawowych.

Tablice zawsze są obiektami, niezależnie od tego, czy zostały zadeklarowane jako tablice jednego z typów podstawowych czy też tablice odwołań do obiektów.

Niemniej jednak można stworzyć obiekt tablicy *zawierającej* dane typów podstawowych. Innymi słowy, obiekt tablicy może posiadać *elementy* typów podstawowych, choć sama tablica *nigdy* nie będzie daną typu podstawowego. Niezależnie od typu swojej zawartości sama tablica zawsze jest obiektem!

Tablice obiektów

Tworzymy tablicę obiektów Pies

- Deklarujemy zmienną będącą tablicą obiektów typu Pies:

```
Pies[] zwierzaki;
```

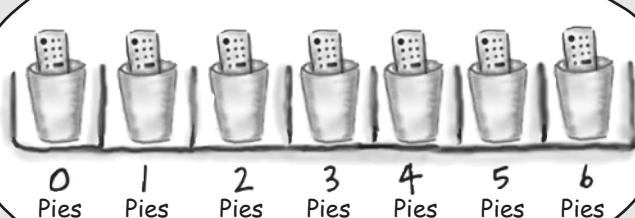
- Tworzymy nową tablicę obiektów Pies składającą się z siedmiu elementów i zapisujemy ją w utworzonej wcześniej zmiennej zwierzaki:

```
zwierzaki = new Pies[7];
```

Czegoś brakuje?

Brakuje obiektów Pies.

Mamy już tablicę odwołań do obiektów Pies, jednak nie mamy samych obiektów!



obiekt tablicy typu Pies (Pies[])

- Tworzymy obiekty Pies i zapisujemy je w elementach tablicy.

Pamiętaj, że elementy tablicy typu Pies są jedynie odwołaniami do obiektów tego typu. Cały czas potrzeba nam samych obiektów!

```
zwierzaki[0] = new Pies();
```

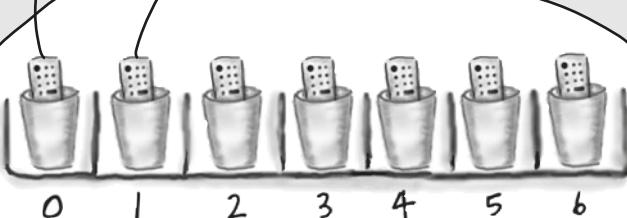
```
zwierzaki[1] = new Pies();
```



Zaostrz ołówek

Jaka jest bieżąca wartość zwierzaki[2]? _____

Jaki kod może sprawić, że element zwierzaki[3] będzie się odwoływać do jednego z dwóch istniejących obiektów Pies?



obiekt tablicy typu Pies (Pies[])



PIES
imie
szczekaj()
jedz()
gonKota()

Java zwraca uwagę na typy.

Kiedy już zadeklarujesz tablicę, nie będziesz mógł umieścić w niej niczego za wyjątkiem danych tego samego typu co tablica.

Na przykład nie można umieścić obiektu Kot w tablicy typu Pies (byłyby bardzo przykro, gdyby ktoś, oczekując, że w tablicy są same Psy, kazał po kolei każdemu z nich szczekać i gdyby nagle okazało się, że gdzieś tam przyciągnął się Kot). Podobnie nie można zapisać liczby typu double w tablicy typu int (pamiętasz, co pisaliśmy o rozsypaniu?). Można jednak zapisać wartość typu byte w tablicy typu int, gdyż byte zawsze będzie pasować do kubka wielkości int. Zjawisko to nosi nazwę **niejawnego rozszerzania**. Bardziej szczegółowo zajmiemy się tym zagadnieniem nieco później, a na razie wystarczy, abyś zapamiętał, że bazując na zadeklarowanym typie tablicy, kompilator nie pozwoli zapisywać w niej danych nieodpowiednich typów.

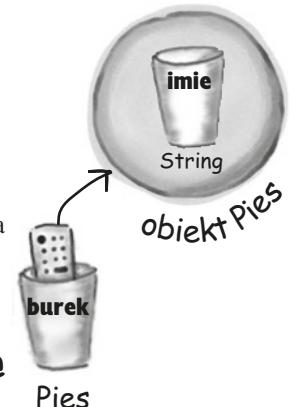
Kontroluj swojego Psa (za pomocą odwołania)

```
Pies burek = new Pies();
```

```
burek.imie = "Burek";
```

Stworzyliśmy obiekt Pies, a następnie użyliśmy operatora kropki wraz ze zmienną referencyjną o nazwie **burek**, w celu uzyskania dostępu do zmiennej określającej imię psa.*

Mozemy użyć odwołania **burek**, aby „nakłonić” psa do wykonania określonych czynności, takich jak szczekaj(), jedz() lub gonKota().



Co się stanie, jeśli Pies będzie zapisany w tablicy?

Wiemy, że za pośrednictwem operatora kropki można uzyskać dostęp do składowych oraz metod obiektu Pies. Ale *na czym* operator ten ma „operować”?

Kiedy Pies jest w tablicy, nie dysponujemy nazwą zmiennej (jak na przykład **burek**). W takim przypadku musimy wykorzystać zapis charakterystyczny dla tablic i nacisnąć przycisk pilota (operator kropki) do obiektu znajdującego się w konkretnym miejscu tablicy (określonym za pomocą indeksu):

```
Pies[] mojePieski = new Pies[3];
```

```
mojePieski[0] = new Pies();
```

```
mojePieski[0].imie = "Burek";
```

```
mojePieski[0].szczekaj();
```

* Owszem. Zdajemy sobie sprawę, że przedstawione rozwiązanie nie jest w pełni zgodne z zasadami hermetyzacji, jednak staramy się nie komplikować niepotrzebnie omawianych zagadnień. Hermetyzacją zajmiemy się w rozdziale 4.

Stosowanie odwołań

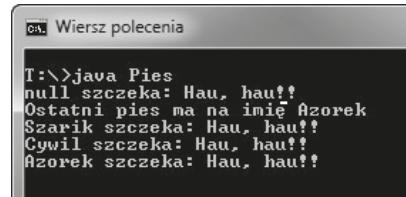
```
class Pies {  
    String imie;  
  
    public static void main (String[] args) {  
        // tworzymy obiekt Pies i używamy go  
        Pies pies1 = new Pies();  
        pies1.szczekaj();  
        pies1.imie = "Azorek";  
  
        // teraz tworzymy tablicę obiektów Pies  
        Pies[] mojePsy = new Pies[3];  
        // i zapisujemy w niej obiekty  
        mojePsy[0] = new Pies();  
        mojePsy[1] = new Pies();  
        mojePsy[2] = pies1;  
  
        // teraz uzyskujemy dostęp do obiektów,  
        // odwołując się do nich przez tablicę  
        mojePsy[0].imie = "Szark";  
        mojePsy[1].imie = "Cywil";  
  
        // Hm.... jak ma na imię pies  
        // z komórki mojePsy[2] ?  
        System.out.print("Ostatni pies ma na imię ");  
        System.out.println(mojePsy[2].imie);  
  
        // A teraz w pętli każemy wszystkim  
        // psom szczekać  
        int x = 0;  
        while (x < mojePsy.length) {  
            mojePsy[x].szczekaj();  
            x = x + 1;  
        }  
    }  
  
    public void szczekaj() {  
        System.out.println(imie + " szczeka: Hau, hau!!");  
    }  
  
    public void jedz() { }  
    public void gonKota() { }  
}
```

Tablice dysponują składową
"length", która określa ilość
elementów w tablicy.

Przykładowy obiekt Pies

PIES
imie
szczekaj()
jedz()
gonKota()

Wyniki:



```
Wiersz polecenia  
T:\>java Pies  
null szczeka: Hau, hau!!  
Ostatni pies ma na imię Azorek  
Szark szczeka: Hau, hau!!  
Cywil szczeka: Hau, hau!!  
Azorek szczeka: Hau, hau!!
```



CELNE SPOSTRZEŻENIA

- Rozróżniamy dwa rodzaje zmiennych — zmienne typów podstawowych oraz odwołania.
- Zmienne zawsze należy deklarować, podając ich nazwę oraz typ.
- Wartościami zmiennych typów podstawowych są ciągi bitów reprezentujące wartości tych zmiennych.
- Wartościami odwołań są z kolei bity reprezentujące sposób, w jaki można się dostać do obiektu przechowywanego na stercie.
- Odwołanie można porównać z pilotem. Użycie operatora kropki wraz z odwołaniem jest jak gdyby naciśnięciem przycisku na pilocie, które ma na celu uzyskanie dostępu do metody obiektu lub jego składowej.
- Kiedy zmienna referencyjna nie wskazuje na żaden obiekt, ma wartość null.
- Tablica zawsze jest obiektem, nawet jeśli została zadeklarowana jako tablica danych jednego z typów podstawowych. Nie istnieje coś takiego jak tablica będąca daną jakiegoś typu podstawowego — istnieją jedynie tablice, w których można przechowywać dane typów podstawowych.



Ćwiczenie

BĄDŹ kompilatorem



Każdy z plików przedstawionych na tej stronie stanowi niezależny, kompletny plik źródłowy. Twoim zadaniem jest stać się kompilatorem i określić, czy przedstawione programy skompilują się czy nie. Jeśli nie można ich skompilować, to jak je poprawić?

A

```
class Ksiazka {
    String tytul;
    String autor;
}

class KsiazkaTester {
    public static void main(String[] args) {

        Ksiazka[] mojeKsiazki = new Ksiazka[3];
        int x = 0;
        mojeKsiazki[0].tytul = "Czterej koderzy i Java";
        mojeKsiazki[1].tytul = "Java nocy letniej";
        mojeKsiazki[2].tytul = "Java. Receptury";
        mojeKsiazki[0].autor = "janek";
        mojeKsiazki[1].autor = "wilhelm";
        mojeKsiazki[2].autor = "ian";

        while (x < 3) {
            System.out.print(mojeKsiazki[x].tytul);
            System.out.print(", autor ");
            System.out.println(mojeKsiazki[x].autor);
            x = x + 1;
        }
    }
}
```

B

```
class Hobbici {

    String imie;

    public static void main(String[] args) {

        Hobbici[] h = new Hobbici[3];
        int z = 0;

        while (z < 4) {
            z = z + 1;
            h[z] = new Hobbici();
            h[z].imie = "Bilbo";
            if (z == 1) {
                h[z].imie = "Frodo";
            }
            if (z == 2) {
                h[z].imie = "Sam";
            }
            System.out.print(h[z].imie + " jest ");
            System.out.println("dobrym imieniem dla hobbita");
        }
    }
}
```

Ćwiczenia: Magnesiki z kodem



Ćwiczenie



Magnesiki z kodem

Działający program Javy został podzielony na fragmenty, zapisany na małych magnesach, które przyczepiono do lodówki. Czy jesteś w stanie złożyć go z powrotem w jedną całość, tak aby wygenerował przedstawione poniżej wyniki? Niektóre nawiasy klamrowe spadły na podłogę i były zbyt małe, aby można je było podnieść, dlatego w razie potrzeby możesz je dodawać!

```
int y = 0;
```

```
ref = indeks[y];
```

```
wyspy[0] = "Bermudy";
wyspy[1] = "Fiji";
wyspy[2] = "Azory";
wyspy[3] = "Kozumel";
```

```
int ref;
while (y < 4) {
```

```
System.out.println(wyspy[ref]);
```

```
indeks[0] = 1;
indeks[1] = 3;
indeks[2] = 0;
indeks[3] = 2;
```

```
String[] wyspy = new String[4];
```

```
System.out.print("wyspa = ");
```

```
int [] indeks = new int[4];
```

```
y = y + 1;
```

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

Wiersz polecenia

```
T:>>java TestTablic
wyspa = Fiji
wyspa = Kozumel
wyspa = Bermudy
wyspa = Azory
```



Zagadkowy basen



Twoim ***zadaniem*** jest wybranie fragmentów kodu z basenu i umieszczenie ich w miejscach kodu oznaczonych podkreśleniami. Każdy fragmentu kodu **możne** być użyty więcej niż raz, a co więcej, nie wszystkie fragmenty zostaną wykorzystane. **Zadanie** polega na stworzeniu klasy, którą będzie można skompilować i która wygeneruje wyniki przedstawione poniżej. Nie daj się zwieść pozorom — ta zagadka jest trudniejsza, niż można by przypuszczać.

Wyniki:

```
ca. Wiersz polecenia

T:\>java Trojkat
trojkat 0, pole = 4.0
trojkat 1, pole = 10.0
trojkat 2, pole = 18.0
trojkat 3, pole = _____
y = _____
```

Pytanie dodatkowe!

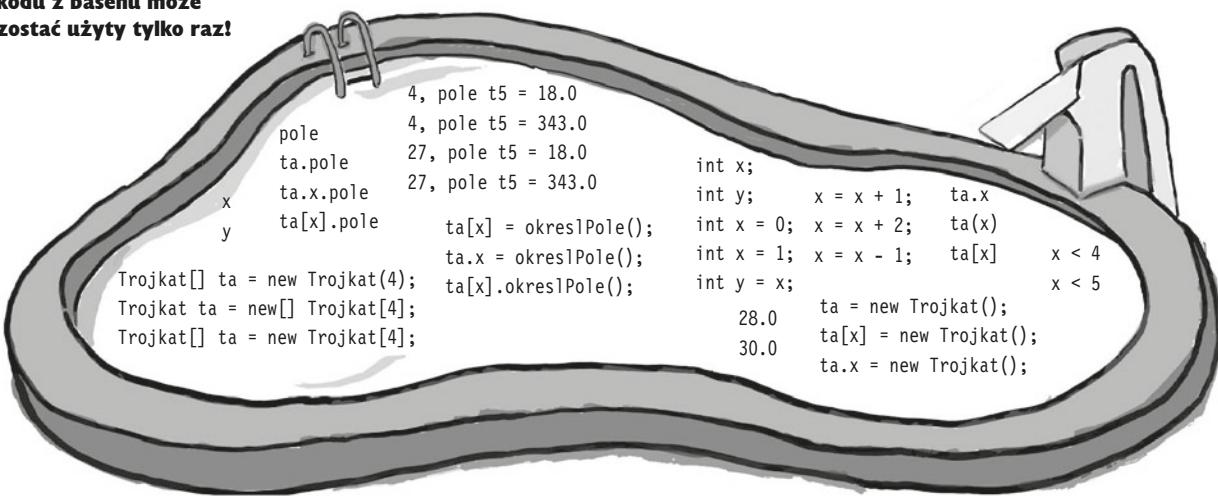
Aby zdobyć dodatkowe punkty, użij fragmentów z basenu, aby uzupełnić brakujące fragmenty wygenerowanych wyników (na powyższym rysunku).

Notatka: Każdy fragment

**kodu z basenu może
zostać użyty tylko raz!**

		4, pole t5 = 18.0
	pole	4, pole t5 = 343.0
	ta.pole	27, pole t5 = 18.0
x	ta.x.pole	27, pole t5 = 343.0
y	ta[x].pole	ta[x] = okresPole(); ta.x = okresPole();
Trojkat[]	ta = new Trojkat[4];	ta[x].okresPole();
Trojkat	ta = new[] Trojkat[4];	
Trojkat[]	ta = new Trojkat[4];	

```
class Trojkat {          Czasem nie używamy oddzielnej
    double pole;          klasy testującej, aby zaoszczędzić
    int wysokosc;         nieco miejsca na stronie.
    int dlugosc;
    public static void main(String[] args) {
        _____
        _____
        while (_____) {
            _____ .wysokosc = (x + 1) * 2;
            _____ .dlugosc = x + 4;
            _____
            System.out.print("trojkat "+x+", pole");
            System.out.println(" = " + _____.pole);
            _____
        }
        _____
        x = 27;
        Trojkat t5 = ta[2];
        ta[2].pole = 343;
        System.out.print("y = " + y);
        System.out.println(", pole t5 = "+ t5.pole);
    }
    void okreslPole() {
        _____ = (wysokosc * dlugosc) / 2;
    }
}
```



Zagadka. Góra problemów



Góra problemów

Z prawej strony przedstawiony został prosty program. Kiedy wykonywanie programu dojdzie do miejsca oznaczonego jako „// dalsze operacje”, będą w nim już utworzone pewne obiekty i zmienne referencyjne. Twoim zadaniem jest określenie, które z tych odwołań wskazują na poszczególne obiekty. Nie wszystkie zmienne będą używane, a na niektóre obiekty może wskazywać więcej niż jedno odwołanie. Narysuj linie łączące odwołania oraz odpowiadające im obiekty.

Podpowiedź: Rozwiążując to zadanie, prawdopodobnie będziesz musiał posłużyć się diagramem, takim jak ten przedstawiony na stronie 87. lub 89. tego rozdziału; no chyba że jesteś znacznie mądrzejszy od nas — wtedy diagram nie będzie Ci potrzebny. Skorzystaj z ołówka, abyś mógł rysować i usuwać linie łączące odwołania i obiekty (strzałki powinny wskazywać kierunek od odwołania do obiektu).

```
class KwizGoraProblemow {  
    int id = 0;  
    public static void main(String[] args) {  
        int x = 0;  
        KwizGoraProblemow[] kwz = new  
        KwizGoraProblemow[5];  
        while (x < 3) {  
            kwz[x] = new KwizGoraProblemow();  
            kwz[x].id = x;  
            x = x + 1;  
        }  
        kwz[3] = kwz[1];  
        kwz[4] = kwz[1];  
        kwz[3] = null;  
        kwz[4] = kwz[0];  
        kwz[0] = kwz[3];  
        kwz[3] = kwz[2];  
        kwz[2] = kwz[0];  
        // dalsze operacje  
    }  
}
```

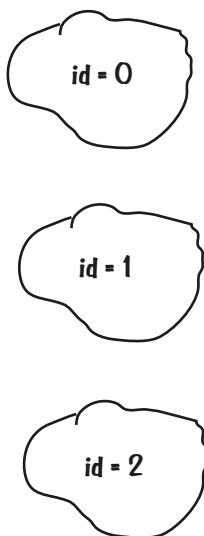
Zmienne referencyjne:



Połącz każde odwołanie z obiektem (lub obiektami).

Nie wszystkie odwołania muszą zostać wykorzystane.

Obiekty KwizGoraProblemow:





Zagadka na pięć minut



Sprawa zaginionych odwołań

To była ciemna i burzliwa noc. Zuzanna wkroczyła do lochu programistów, jak gdyby to miejsce było jej własnością. Doskonale wiedziała, że o tej porze wszyscy programiści będą jeszcze ciężko pracować, a potrzebowała pomocy. Potrzebowała nowej metody, którą trzeba było dodać do kluczowej klasy wchodzącej w skład mobilnej aplikacji, jaka miała być zainstalowana w obslugiwany przez Javę telefonie komórkowym klienta. Wielkość sterty w telefonach komórkowych jest tak skąpa jak bluzka Zuzanny, i każdy o tym wiedział. Gdy Zuzanna szła w kierunku białej tablicy, ochrypte szepły wypełniające pomieszczenie zmieniły się w głuchą ciszę. Dziewczyna narysowała na tablicy krótki schemat funkcjonalny metody i powoli przyjrzała się mężczyznom zgromadzonym w pomieszczeniu. — Chłopcy — powiedziała cichym, kokieteryjnym głosem — zbliża się czas odebrania nagrody. Ten z was, który napisze wersję metody wykorzystującą najmniejszą możliwą ilość pamięci, poleci ze mną jutro do klienta na Maui... pomóc mi zainstalować aplikację, oczywiście.

Następnego ranka Zuzanna wkroczyła do lochu ubrana w króciutką sukienkę w stylu Aloha. — Panowie — uśmiechnęła się niewinnie — samolot odlatuje za kilka godzin, pokażcie, co dla mnie macie! Pierwszy wstał Robert; gdy zaczął rysować na tablicy projekt swojego rozwiązania Zuzanna powiedziała: — Robert, do rzeczy, pokaż mi, jak rozwiązałeś problem aktualizacji listy kontaktów. Robert szybko napisał na tablicy krótki fragment kodu:

```
Kontakt [] kn = new Kontakt[10];
while (x < 10) { // tworzymy 10 obiektów kontaktów
    kn[x] = new Kontakt();
    x = x + 1;
}
// aktualizujemy listę kontaktów posługując się tablica kn
```

— Zuza, wiem, że nie dysponujemy dużą ilością pamięci, ale według twojej specyfikacji musimy mieć dostęp do każdego z dziesięciu dozwolonych kontaktów; to był najlepszy schemat, jaki udało mi się wymyślić — powiedział Robert. Następny był Leszek. Wyobrażając już sobie koktaile sączone z orzecha kokosowego prychnął: — Stary, Twoje rozwiązanie jest chyba nieco toporne — zerknij na to cudeńko — i napisał na tablicy:

```
Kontakt odwk;
while (x < 10) { // tworzymy 10 odwołań do obiektów
    odwk = new Kontakt();
    x = x + 1;
}
// wykonujemy złożoną aktualizację listy kontaktów, posługując się
// odwołaniem odwk
```

— Zaoszczędziłem kilka odwołań w naszej cennej pamięci, Robciu — zaśmiała się pogardliwie.
— Zatem chyba możesz się pożegnać ze swoim kremem do opalania. Nie tak szybko Leszku — powiedziała Zuzanna. — Fakt, oszczędziłeś trochę pamięci, ale na Maui poleci ze mną Robert.

Dlaczego Zuzanna wybrała metodę Roberta, a nie Leszka, skoro to właśnie metoda Leszka wykorzystywała mniejszą ilość pamięci?

Rozwiązań zadań



Ćwiczenie rozwiązanie

Magnesiki z kodem

```
class TestTablic {  
    public static void main(String[] args){  
        int [] indeks = new int[4];  
        indeks[0] = 1;  
        indeks[1] = 3;  
        indeks[2] = 0;  
        indeks[3] = 2;  
        String[] wyspy = new String[4];  
        wyspy[0] = "Bermudy";  
        wyspy[1] = "Fiji";  
        wyspy[2] = "Azory";  
        wyspy[3] = "Kozumel";  
        int y = 0;  
        int ref;  
        while (y < 4) {  
            ref = indeks[y];  
            System.out.print("wyspa = ");  
            System.out.println(wyspy[ref]);  
            y = y + 1;  
        }  
    }  
}
```

```
Wiersz polecenia  
T:\>java TestTablic  
wyspa = Fiji  
wyspa = Kozumel  
wyspa = Bermudy  
wyspa = Azory
```

```
class Ksiazka {  
    String tytul;  
    String autor;  
}  
  
class KsiazkaTester {  
    public static void main(String[] args) {  
  
        Ksiazka[] mojeKsiazki = new Ksiazka[3];  
        int x = 0;  
        mojeKsiazki[0] = new Ksiazka(); Pamiętaj! Trzeba utworzyć obiekty Ksiazka!  
        mojeKsiazki[1] = new Ksiazka();  
        mojeKsiazki[2] = new Ksiazka();  
        mojeKsiazki[0].tytul = "Czterej koderzy i Java";  
        mojeKsiazki[1].tytul = "Java nocą letniej";  
        mojeKsiazki[2].tytul = "Java. Receptury";  
        mojeKsiazki[0].autor = "janek";  
        mojeKsiazki[1].autor = "wilhelm";  
        mojeKsiazki[2].autor = "ian";  
        while (x < 3) {  
            System.out.print(mojeKsiazki[x].tytul);  
            System.out.print(", autor ");  
            System.out.println(mojeKsiazki[x].autor);  
            x = x + 1;  
        }  
    }  
}
```

```
class Hobbici {  
    String imie;  
  
    public static void main(String[] args) {  
  
        Hobbici[] h = new Hobbici[3];  
        int z = -1;  
        while (z < 2) { Pamiętaj, że indeksy elementów tablic zaczynają się od wartości 0!  
            z = z + 1;  
            h[z] = new Hobbici();  
            h[z].imie = "Bilbo";  
            if (z == 1) {  
                h[z].imie = "Frodo";  
            }  
            if (z == 2) {  
                h[z].imie = "Sam";  
            }  
            System.out.print(h[z].imie + " jest ");  
            System.out.println("dobrym imieniem dla hobbita");  
        }  
    }  
}
```

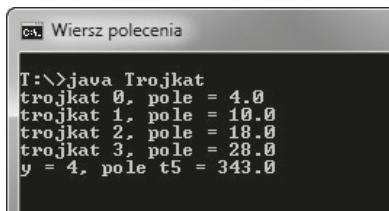


Rozwiążanie zagadki

```

class Trojkat {
    double pole;
    int wysokosc;
    int dlugosc;
    public static void main(String[] args) {
        int x = 0;
        Trojkat[] ta = new Trojkat[4];
        while (x < 4) {
            ta[x] = new Trojkat();
            ta[x].wysokosc = (x + 1) * 2;
            ta[x].dlugosc = x + 4;
            ta[x].okreslPole();
            System.out.print("trojkat "+x+", pole");
            System.out.println(" = " + ta[x].pole);
            x = x + 1;
        }
        int y = x;
        x = 27;
        Trojkat t5 = ta[2];
        ta[2].pole = 343;
        System.out.print("y = " + y);
        System.out.println(", pole t5 = "+ t5.pole);
    }
    void okreslPole() {
        pole = (wysokosc * dlugosc) / 2;
    }
}

```



Sprawa zaginionych odwołań

Zuzanna zauważała, że rozwiązanie Leszka ma poważny błąd. To prawda, że użył on mniejszej ilości odwołań niż Robert, jednak metoda dawała możliwości dostępu jedynie do ostatniego utworzonego obiektu Kontakt. Podczas każdej iteracji pętli Leszek zapisywał nowy obiekt Kontakt w jednej i tej samej zmiennej, a zatem wcześniej utworzony obiekt, do którego odwołanie było w tej zmiennej przechowywane, zostawał porzucony na stercie, a tym samym stawał się *nieosiągalny*. Bez możliwości uzyskania dostępu do dziewięciu spośród dziesięciu utworzonych obiektów metoda Leszka stawała się bezużyteczna.

Aplikacja odniosła wielki sukces, a klient zafundował Zuzannie i Robertowi dodatkowy tydzień na Hawajach. Chcielibyśmy Ci powiedzieć, że po zakończeniu lektury niniejszej książki też będziesz miał takie okazje.

Zmienne referencyjne:



`kwz[0]`



`kwz[1]`



`kwz[2]`

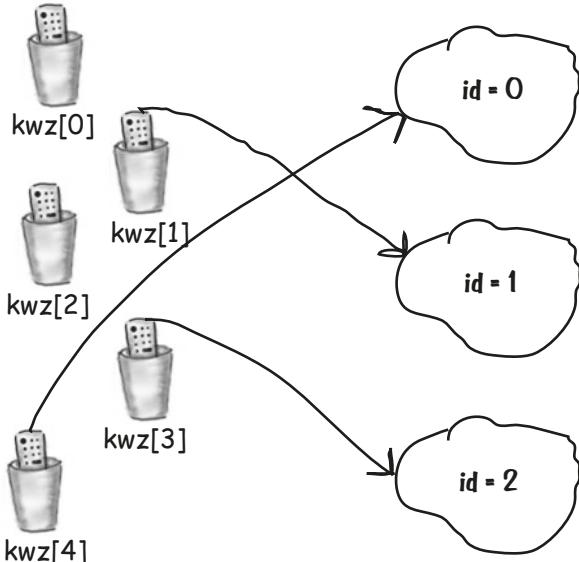


`kwz[3]`



`kwz[4]`

Obiekty KwizGoraProblemow:



4. Metody wykorzystują składowe

Jak działają obiekty?



Stan wpływa na działanie, a działanie wpływa na stan. Wiemy już, że obiekty mają **stan** oraz **działanie**, które to aspekty obiektów są odpowiednio reprezentowane przez **składowe** oraz **metody**. Jednak do tej pory nie zajmowaliśmy się zagadnieniem, w jaki sposób stan oraz działanie obiektu są ze sobą *powiązane*. Wiemy już, że każda kopia klasy (każdy obiekt konkretnego typu) może mieć własne, unikalne wartości składowych. Pies A może się wabić „Azor” (składowa *imie*) i ważyć 25 kilogramów (składowa *waga*). Z kolei Pies B może się wabić „Kiler” i ważyć 3 kilogramy. Jeśli klasa Pies będzie mieć metodę *halasuj()*, to czy nie uważasz, że 25-kilogramowe psisko będzie szczekać inaczej niż trzykilogramowy piesek? (Zakładając w ogóle, że te wysokie piski można uznać za szczekanie.) Na szczęście właśnie to jest podstawową cechą obiektów — *działania* konkretnego obiektu operują na jego *stanie*. Innymi słowy, **metody wykorzystują wartości składowych**. Na przykład, „jeśli pies waży mniej niż 7 kilogramów, szczekaj piskliwie, w przeciwnym razie...” bądź też: „powiększ wagę o 2 kilogramy”. **A zatem — pozmieniamy stan.**

Obiekty mają stan i działanie

Pamiętaj! Klasa opisuje to, co obiekt wie, oraz to, co robi

Klasa jest szablonem obiektów. Tworząc klasę, opisujesz jednocześnie, jak JVM powinna tworzyć obiekty tego typu. Wiesz już, że każdy obiekt konkretnego typu może mieć własne wartości *składowych*. A co z metodami?

Czy w każdym obiekcie danego typu metody mogą działać w odmienny sposób?

Cóż... Coś w tym stylu.*

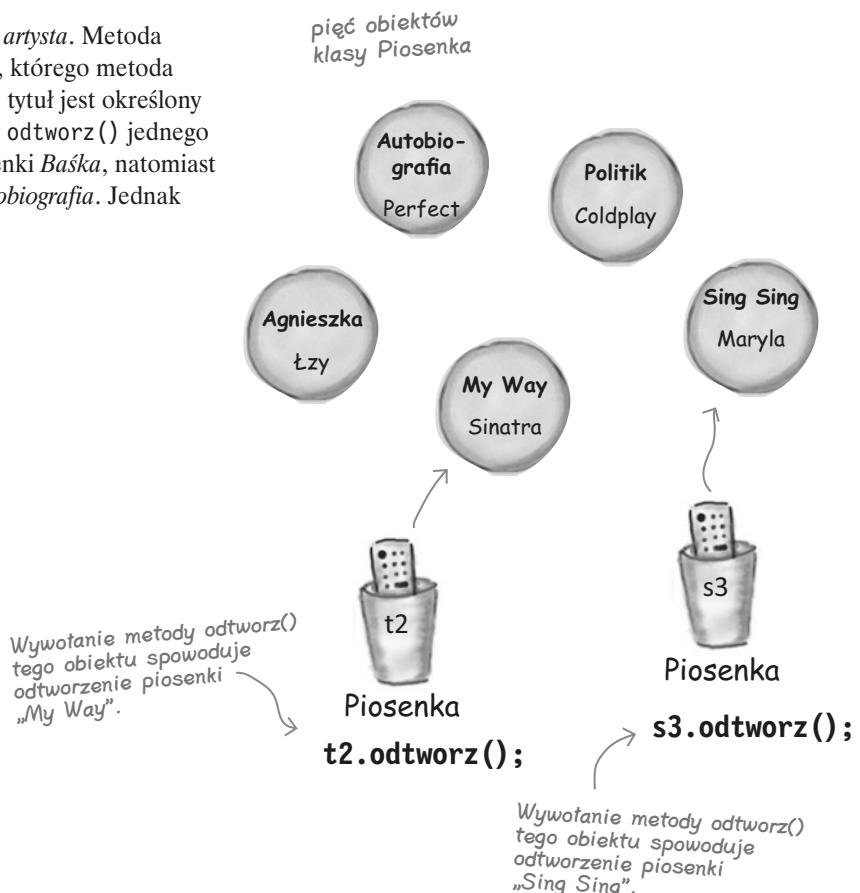
Każdy obiekt danej klasy ma te same metody, jednak same metody mogą działać w różny sposób, w zależności od wartości składowych.

Klasa *Piosenka* ma dwie składowe: *tytuł* oraz *artysta*. Metoda *odtworz()* odtwarza piosenkę, jednak obiekt, którego metoda zostanie wywołana, odtworzy piosenkę, której tytuł jest określony w składowej *tytuł*. A zatem wywołanie metody *odtworz()* jednego obiektu może spowodować odtworzenie piosenki *Baśka*, natomiast w przypadku innego obiektu może to być *Autobiografia*. Jednak w obu przypadkach kod metody jest taki sam.

```
void odtworz() {  
    odtwarzacz.odtworzUtwor(tytul);  
}
```

```
Piosenka t2 = new Piosenka();  
t2.setArtysta("Maryla");  
t2.setTytul("Sing Sing");  
  
Piosenka s3 = new Piosenka();  
s3.setArtysta("Łzy");  
s3.setTytul("Agnieszka");
```

Piosenka	
składowe (stan)	tytuł artysta
metody (działanie)	setTytul() setArtysta() odtworz()



* Tak, to kolejna jasna i precyzyjna odpowiedź.

Wielkość ma wpływ na sposób szczekania

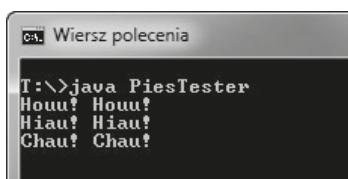
Sposób szczekania małego psa różni się od sposobu szczekania dużego.

Klasa Pies dysponuje składową o nazwie wielkosc, która jest wykorzystywana w metodzie szczekaj() do podejmowania decyzji, jaki dźwięk należy wydać.

```
class Pies {
    int wielkosc;
    String imie;
    void szczekaj() {
        if (wielkosc > 23) {
            System.out.println("Houu! Houu!");
        } else if (wielkosc > 6) {
            System.out.println("Chau! Chau!");
        } else {
            System.out.println("Hiau! Hiau!");
        }
    }
}
```



```
class PiesTester {
    public static void main(String[] args) {
        Pies pierwszy = new Pies();
        pierwszy.wielkosc = 40;
        Pies drugi = new Pies();
        drugi.wielkosc = 2;
        Pies trzeci = new Pies();
        trzeci.wielkosc = 8;
        pierwszy.szczerkaj();
        drugi.szczerkaj();
        trzeci.szczerkaj();
    }
}
```



Do metod można przekazywać informacje

Jak można było oczekwać od dobrego języka programowania, istnieje możliwość przekazywania wartości do metod. Na przykład mógłbyś chcieć, stosując poniższe wywołanie, poinformować obiekt Pies, ile razy należy zaszczekać:

`d.szczekaj(3);`

Zależnie od posiadanych doświadczeń programistycznych i własnych preferencji, wartości przekazywane do metod można nazywać *argumentami* lub *parametrami*. Choć pomiędzy tymi terminami *istnieją formalne różnice*, które na pewno podałby osoby noszące fartuchy laboratoryjne, jednak one niemal na pewno nie będą czytać takiej książki jak ta, a my mamy ważniejsze zadania niż spieranie się o terminologię. Zatem możesz nazywać te wartości tak, jak Ci pasuje (argumentami, pączkami, klębkami itp.), my jednak zrobimy w następujący sposób:

Metoda używa parametrów. Kod wywołujący metodę przekazuje do niej argumenty.

Argumenty są elementami przekazywanymi do metody. **Argument** (wartość, taka jak 2, „Napis”, odwołanie do obiektu Pies), zostaje zapisany w..., jak to było?, **parametrze**. Parametr nie jest niczym innym, niż zwyczajną zmienną lokalną. Zmienną, która posiada swój typ, nazwę i która może być wykorzystywana w kodzie metody.

Jednak najważniejsze jest to, iż **jeśli metoda ma jakieś parametry, to koniecznie trzeba coś do niej przekazać**. A to coś, musi być wartością odpowiedniego typu.

- 1 Wywołujemy metodę `szczekaj()`, posługując się odwołaniem do obiektu Pies i przekazujemy do niej wartość 3 (jako argument jej wywołania).

```
void szczekaj(int iloscSzczekniec) {  
    while (iloscSzczekniec > 0) {  
        System.out.println("hauu");  
        iloscSzczekniec = iloscSzczekniec - 1;  
    }  
}
```

`Pies p = new Pies();`
`p.szczekaj(3);`

- 2 Bity reprezentujące wartość całkowitą 3 są przekazywane do metody `szczekaj()`.

- 3 Bity są umieszczane w parametrze `iloscsSzczekniec` (typu int).

- 4 Parametr `iloscsSzczekniec` jest używany jako zmienna w kodzie metody.

Metoda może coś zwrócić

Metody mogą zwracać wartości. W deklaracjach metod można określić typ wyniku, jednak jak do tej pory wszystkie tworzone przez nas metody były typu **void**, co oznacza, że nie zwracały żadnych wyników.

```
void idz() {  
}
```

Można jednak zadeklarować metodę, która będzie zwracać do kodu wywołującego wartości określonego typu. Oto przykład takiej metody:

```
int podajTajnyNumer() {  
    return 42;  
}
```

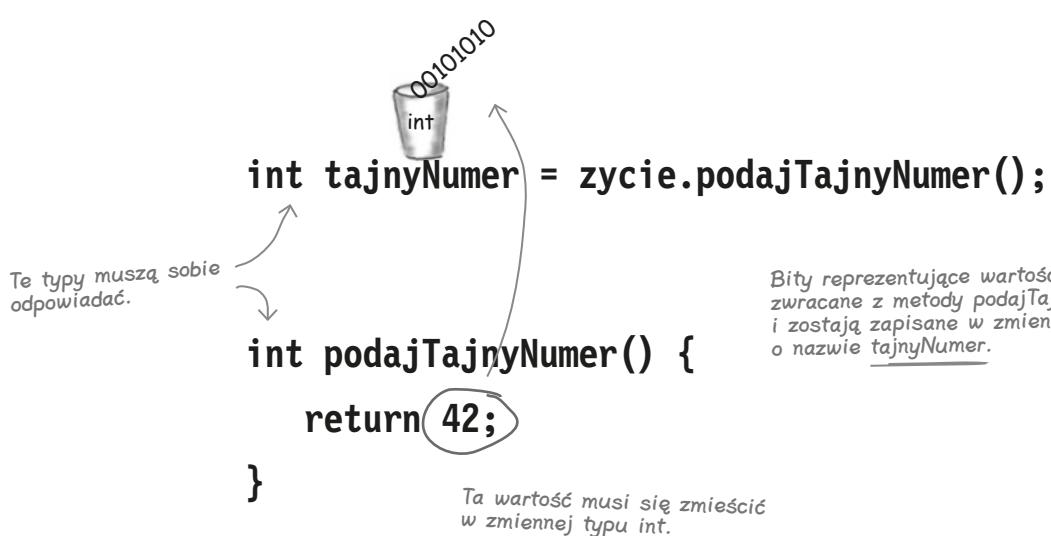
Jeśli w deklaracji metody zostanie określona wartość wynikowa, to zwrócenie wartości podanego typu jest *konieczne!* (Można także zwrócić wartość, która jest *zgodna* z wartością zadeklarowanego typu. Więcej informacji na temat tego zagadnienia podamy przy okazji omawiania zagadnień związanych z polimorfizmem, w rozdziałach 7. i 8.)

**Radzę Ci! Jeśli obiecałeś,
że metoda będzie coś zwracać,
to lepiej to zwracaj.**

Urocze... choć niezupełnie tego się spodziewałam.



Kompilator nie pozwoli zwrócić wartości niewłaściwego typu.



Do metody można przekazać więcej niż jedną informację

Metody mogą posiadać wiele parametrów. W deklaracji metody poszczególne parametry należy oddzielać od siebie przecinkami, podobnie jak argumenty w jej wywołaniu.

Najważniejsze jest jednak to, że jeśli metoda ma jakieś parametry, to koniecznie trzeba przekazać do niej argumenty odpowiednich typów i w odpowiedniej kolejności.

Wywołanie metody z dwoma parametrami i przekazanie do niej dwóch argumentów.

```
void idz() {  
    Tester t = new Tester();  
    t.dwaParametry(12, 34);  
}  
  
void dwaParametry(int x, int y) {  
    int z = x + y;  
    System.out.println("Suma wynosi: " + z);  
}
```

Przekazywane argumenty są zapisywane w takiej samej kolejności, w jakiej zostały podane. Pierwszy argument trafia do pierwszego parametru, drugi argument do drugiego parametru i tak dalej.

Do metody można przekazywać wartości zmiennych, o ile typy tych zmiennych odpowiadają typom parametrów.

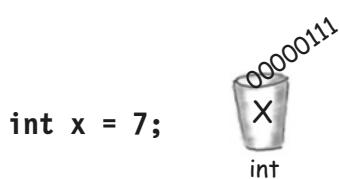
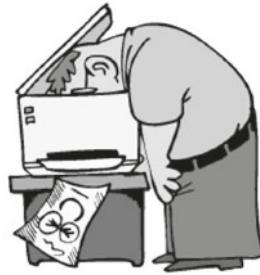
```
void idz() {  
    Tester t = new Tester();  
    int arg1 = 7;  
    int arg2 = 3;  
    t.dwaParametry(arg1, arg2);  
}  
  
void dwaParametry(int x, int y) {  
    int z = x + y;  
    System.out.println("Suma wynosi: " + z);  
}
```

Wartości zmiennych arg1 i arg2 są umieszczane, odpowiednio, w parametrach x oraz y. A zatem, aktualnie, bity w parametrze x są takie same jak bity w zmiennej arg1 (są to bity odpowiadające wartości całkowitej 7), a bity w y są takie same jak bity w zmiennej arg2.

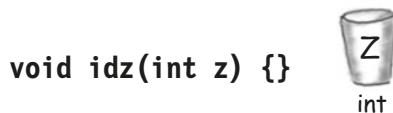
Jaka jest wartość z? Ten sam wynik można by uzyskać, dodając zmienne arg1 i arg2, w tym samym czasie, gdy ich wartości zostają przekazane do metody dwaParametry.

Java przekazuje argumenty przez wartość.

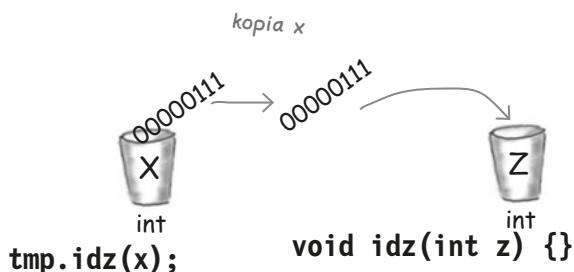
To oznacza, że przekazywana jest kopia.



- 1 Deklarujemy zmienną typu int i przypisujemy jej wartość „7”. Ciąg bitów reprezentujący wartość 7 jest zapisywany w zmiennej o nazwie x.

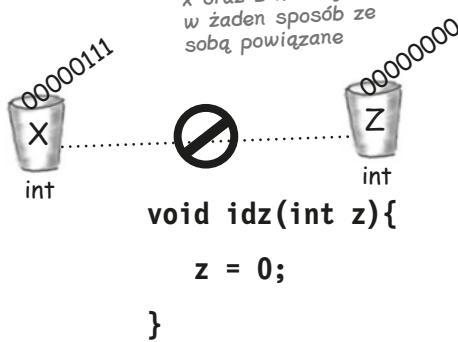


- 2 Deklarujemy metodę posiadającą parametr typu int o nazwie z.



- 3 Wywołujemy metodę idz(), przekazując zmienną x jako argument jej wywołania. Bity stanowiące zawartość zmiennej x są kopiowane, a ich kopia jest zapisywana w parametrze z.

Wartość zmiennej x nie zmieni się, nawet jeśli z ulegnie zmianie.



- 4 Wewnątrz metody zmieniamy wartość parametru z. Wartość x nie ulega zmianie! Argument przekazany do parametru z był jedynie kopią zmiennej x. Metoda nie może zmienić ciągu bitów stanowiącego zawartość zmiennej x.

Argumenty i wartości wynikowe

Nie istnieja
głupie pytania

P: Co się dzieje, jeśli argument, który chcemy przekazać, jest obiektem, a nie wartością jednego z typów podstawowych?

O: Więcej na ten temat dowiesz się w kolejnych rozdziałach, jednak już znasz odpowiedź na to pytanie. Java wszystko przekazuje przez wartość. **Wszystko.** Jednak... wartość oznacza ciąg bitów przechowywany w zmiennej. Pamiętaj, że obiekty nie są zapisywane w zmiennych; zmienna jest jedynie pilotem do obiektu — odwołaniem do niego. A zatem, przekazując w wywoaniu metody odwołanie do obiektu, przekazujemy do niej kopię pilota. Nie martw się — później będziemy mieli znacznie więcej do powiedzenia na ten temat.

P: Czy w deklaracji metody można zaznaczyć, że zwraca ona wartości różnych typów? Bądź też czy istnieje sposób zwracania więcej niż jednej wartości?

O: Coś w tym stylu. W deklaracji metody można podać tylko jedną wartość wynikową. ALE... jeśli chcesz zwrócić, na przykład, trzy liczby całkowite, to zadeklarowana wartość wynikowa może być tablicą typu int. Wystarczy zapisać liczby całkowite do tablicy i zwrócić ją jako wynik metody. Zwracanie kilku wartości różnych typów jest nieco bardziej złożone; problemem tym zajmiemy się w dalszej części książki, przy okazji omawiania klasy ArrayList.

P: Czy trzeba zwracać wartość dokładnie tego samego typu, który został podany w deklaracji metody?

O: Można zwracać dowolne wartości, które mogą być w niejawnym sposobie rozszerzone do danego typu. A zatem jeśli oczekiwana jest liczba całkowita typu int, można przekazać bajt — czyli wartość typu byte. Kod wywołujący metodę nie zwróci na to uwagi, gdyż bajt bez problemu będzie można zapisać w zmiennej całkowitej, która zostanie użyta do przechowania wyniku zwróconego przez metodę. W przypadku gdy zadeklarowany typ jest mniejszy od typu wartości, którą chcesz zwrócić, trzeba będzie użyć jawnego rzutowania typów.

P: Czy trzeba coś zrobić z wartością wynikową zwróconą przez metodę? Czy można ją zignorować?

O: Java nie wymaga jakiegokolwiek potwierdzania faktu odebrania wartości wynikowej. Może się zdarzyć, że będziesz chciał wywołać metodę zwracającą jakąś wartość (czyli daną typu innego niż void), nawet jeśli sama zwrócona wartość nie ma żadnego znaczenia. W takim przypadku metoda jest wywoływana ze względu na czynności, jakie są wykonywane *wewnątrz* niej, a nie ze względu na wartość wynikową. W Javie nie ma obowiązku przypisywania wartości wynikowej do jakiegokolwiek zmiennej ani wykorzystywania jej w dalszej części kodu.



Przypomnienie: Java zwraca uwagę na typy!

Nie można zwracać danej typu Żyrafa, jeśli zadeklarowano wartość wynikową typu Królik. Ta sama zasada dotyczy także parametrów. Nie można przekazać Żyrafy do metody, która wymaga przekazania Królika.



CELNE SPOSTRZEŻENIA

- Klasa definiuje to, co obiekt wie, oraz to, co robi.
- To, co obiekt wie, jest przechowywane w **składowych** (jest to stan obiektu).
- Czynności, jakie obiekt wykonuje, to **metody** (stanowiące działanie obiektu).
- Metody wykorzystują składowe, dzięki czemu obiekty tego samego typu mogą zachowywać się w różny sposób.
- Metody mogą mieć parametry, co oznacza, że można do nich przekazać jedną lub kilka wartości.
- Ilość i typy wartości przekazywanych do metody muszą odpowiadać ilości i typom zadeklarowanych parametrów tej metody.
- Wartości przekazywane do metody oraz zwracane przez nią mogą być niejawnie rozszerzane do wartości większych typów lub jawnie rzutowane na wartości mniejszych typów.
- Wartości przekazywane w wywołaniu metody mogą być literałami (takimi jak 2, 'c' i tak dalej) lub zmiennymi, których typ odpowiada zadeklarowanemu typowi parametru (na przykład x, gdzie x jest zmienną typu int). (Argumenty mogą być także danymi innych typów, jednak tymi zagadnieniami zajmiemy się później).
- Metoda musi deklarować jakąś wartość wynikową. Typ void oznacza, że metoda nie zwraca żadnej wartości.
- Jeśli w deklaracji metody podano, że zwraca ona wartość typu innego niż void, to *musi* ona zwrócić wartość, której typ będzie zgodny z zadeklarowanym typem wartości wynikowej.

Ciekawe rozwiązania wykorzystujące parametry i wartości wynikowe

Teraz, kiedy już wiesz, jak działają parametry oraz wartości wynikowe, nadszedł czas, aby je wykorzystać, a konkretnie — użyć ich przy tworzeniu metod **ustawiających** oraz **zwracających**. Jeśli chciałbyś podejść do tego w sposób formalny, być może będziesz wolał nazywać je metodami. Ale, według nas, to strata czasu. Poza tym określenia: metoda zwracająca i metoda ustawiająca doskonale odpowiadają konwencji nazewnicyowej używanej w Javie, dlatego też będziemy ich używać.

Metody zwracające i ustawiające, jak sama nazwa wskazuje, pozwalają, odpowiednio, na *zwracanie* i *ustawianie danych*. Zazwyczaj operują one na wartościach składowych. Jedynym celem istnienia metody zwracającej jest zwrócenie, jako wyniku, wartości konkretnej składowej, na jakiej dana metoda operuje. W takim razie nie zaskoczy Cię zapewne fakt, że metoda ustawiająca istnieje tylko po to, aby skorzystać z okazji pobrania wartości argumentu i zapisania jej w odpowiedniej składowej.

```
class GitaraElektryczna {
    String rodzaj;
    int iloscKonwerterow;
    boolean uzywanaPrzezGwiazde;

    String getRodzaj() {
        return rodzaj;
    }

    void setRodzaj(String rodzajGitary) {
        rodzaj = rodzajGitary;
    }

    int getIloscKonwerterow() {
        return iloscKonwerterow;
    }

    void setIloscKonwerterow(int ilosc) {
        iloscKonwerterow = ilosc;
    }

    boolean getUzywanaPrzezGwiazde() {
        return uzywanaPrzezGwiazde;
    }

    void setUzywanaPrzezGwiazde(boolean takCzyNie) {
        uzywanaPrzezGwiazde = takCzyNie;
    }
}
```

GitaraElektryczna
rodzaj
ilosckKonwerterow
uzywanaPrzezGwiazde
getRodzaj()
setRodzaj()
getIloscKonwerterow()
setIloscKonwerterow()
getUzywanaPrzezGwiazde()
setUzywanaPrzezGwiazde()

Notatka: Wykorzystanie tej konwencji nazewnicyowej będzie oznaczać, że postępujesz zgodnie z ważnym standardem języka Java!



Hermetyzacja

**Jeśli nie będziesz jej używać,
narażasz się na upokorzenie i śmieszność.**

Aż do tego niezwykle ważnego momentu popełnialiśmy jedną z największych gaf możliwych w świecie programowania obiektowego. I bynajmniej nie mówimy tu o żadnym niewielkim wykroczeniu, lecz o prawdziwej gafie przez wielkie „G”.

Cóż to za haniebne przestępstwo?

Obnażanie danych!

Oto co robiliśmy — bezmyślnie pisaliśmy klasy, w których nasze dane były widoczne jak na dłoni, tak że *każdy* mógł je odczytać, a nawet zmienić.

Być może już poznałeś to uczucie delikatnego niepokoju, które rodzi świadomość tego, że nasze składowe są dostępne dla *każdego* i bez żadnych ograniczeń.

Oznacza to, że można się do nich odwołać, wykorzystując operator kropki, jak w poniższym przykładzie:

kot.wysokosc = 27;

Przeanalizuj możliwość wykorzystania naszego pilota do wprowadzenia bezpośredniej zmiany składowej określającej wielkość obiektu Kot. Odwołanie (pilot do obiektu) w rękach nieodpowiedniej osoby może być bardzo niebezpieczną bronią. No bo cóż może nas zabezpieczyć przed wykonaniem następującej instrukcji:

kot.wysokosc = 0; ← Rany! Nie możemy do tego dopuścić!

To byłaby prawdziwa tragedia. Musimy zatem stworzyć metody ustawiające dla wszystkich składowych obiektu oraz znaleźć jakiś sposób, aby zagwarantować, że inne fragmenty kodu będą określać wartości składowych za pośrednictwem tych metod, a nie w sposób bezpośredni.



Wymuszając użycie metody ustawiającej, możemy zabezpieczyć naszego kota przed nieoczekiwaniymi i niedozwolonymi zmianami wielkości.

```
public void setWysokosc(int w) {  
    if (w > 9) {  
        wysokosc = w;  
    }  
}
```

Umieściliśmy tu warunek, który zagwarantuje, że nasz kot nie będzie zbyt mały.

Ukryj dane

Właśnie *tak* łatwo można zmienić implementację, która wprost doprasza się o podanie błędnych informacji w rozwiążaniu, które nie tylko chroni dane, *lecz dodatkowo* zapewnia nam prawo do późniejszej zmiany implementacji.

W porządku, zatem jak można *ukryć* dane? Przy użyciu modyfikatorów dostępu **public** oraz **private**. Pierwszy z nich — **public** — już znasz, używasz go w każdej metodzie **main()**.

Poniżej przedstawiona została *początkowa*, praktyczna reguła hermetyzacji (przy czym obowiązują wszystkie standardowe zasady dotyczące takich praktycznych reguł) — wszystkie składowe należy deklarować jako *privatne*, a kontrola dostępu powinna być realizowana za pomocą *publicznych* metod ustawiających i zwracających. Kiedy zdobędziesz nieco więcej doświadczenia w projektowaniu i kodowaniu w Javie, zapewne będziesz stosował inne rozwiązania, jednak ta reguła zapewni Ci bezpieczeństwo już teraz.

Składowe obiektów należy oznać jako **privatne**.

Metody ustawiające i zwracające powinny być **publiczne**.

„To smutne, lecz to, że Bronek zapomniał o hermetyzacji swojej klasy Kot, przyniósło opłakane skutki — jego Kot stał się płaski jak naleśnik.”

(zasłyszane przy maszynie z napojami orzeźwiającymi).



Java bez tajemnic

Temat wywiadu w tym tygodniu:

Obiekt ujawnia całą prawdę o hermetyzacji

Autorzy: A zatem, co jest najważniejsze w hermetyzacji?

Obiekt: Cóż, czy znacie ten sen, w którym prowadzicie wykład dla 500 osób i nagle zdajecie sobie sprawę, że jesteście *nadzy*?

Autorzy: O tak, znamy go. Jest gdzieś tam — w naszych głowach — wraz z koszmarem o spadaniu w przepaść i... o nie, nie chcemy o tym mówić. W porządku, zatem czujesz się nagi. Ale czy oprócz nieznacznego obnażenia są jeszcze jakieś inne bezpieczeństwa?

Obiekt: Czy są jakieś bezpieczeństwa? Czy są jakieś *niebezpieczeństwa*? (Obiekt zaczyna się śmiać). Hej, czy wszystkie inne kopie to słyszały? *Oni się pytają, czy są jakieś bezpieczeństwa.* (Obiekt spada z krzesła i tarza po podłodze, śmiejąc się spazmatycznie).

Autorzy: Co w tym takiego śmiesznego? To chyba całkiem rozsądne pytanie.

Obiekt: Dobra, już wyjaśniam. To... (Obiekt ponownie, w niekontrolowany sposób wybucha śmiechem).

Autorzy: Czy możemy Ci jakoś pomóc? Może wody?

Obiekt: Rany! Chłopie! Nie dzięki... już dobrze. Teraz będę poważny. Tylko wezmę głęboki oddech. W porządku — możemy kontynuować.

Autorzy: A zatem, przed czym chroni Cię hermetyzacja?

Obiekt: Hermetyzacja otacza moje składowe polem silowym, dzięki któremu nikt nie może przypisać im, ujmijmy to delikatnie, *niewłaściwych* danych.

Autorzy: Czy możesz podać jakiś przykład?

Obiekt: To nie wymaga wielkiej filozofii. W przypadku większości składowych istnieją pewne założenia dotyczące granic ich wartości. Na przykład wyobraźcie sobie wszystkie potencjalne awarie, które mogłyby się wydarzyć, gdyby możliwe było użycie wartości mniejszych od zera. Numery pokoi w biurze. Prędkości samolotów. Waga sztangi. Daty urodzin. Numery telefonów komórkowych. Zasilanie kuchenki mikrofalowej.

Autorzy: Rozumiem, co masz na myśli. A zatem w jaki sposób hermetyzacja pomaga Ci określić te granice?

Obiekt: Poprzez wymuszenie, aby inne fragmenty kodu określały wartości składowych za pośrednictwem metod ustawiających. Dzięki temu metoda ustawiająca jest w stanie sprawdzić parametr i określić, czy jego wartość można zapisać w składowej. Może metoda ta stwierdzi, że wartość jest zła i jej nie przypisz, a może zgłosi jakiś wyjątek (na przykład, gdy w aplikacji operującej na kartach kredytowych pojawi się pusty numer ubezpieczenia społecznego), a może zaokrągli wartość parametru do najbliższej dopuszczalnej wartości. Najważniejsze jest to, że w metodach ustawiających możecie zrobić wszystko, natomiast jeśli składowe są publiczne — nie możecie zrobić *nic*.

Autorzy: Ale czasami można się spotkać z metodami ustawiającymi, które po prostu zapisują parametr w składowej, bez jakiegokolwiek kontroli jego wartości. Jeśli masz składową, której wartości nie są w żaden sposób ograniczane, to czy stosowanie metody ustawiającej nie jest niepotrzebnym narzutem czasowym? Czynnikiem pogarszającym efektywność działania?

Obiekt: Jeśli chodzi o metody ustawiające (jak również zwracające) najważniejsze jest to, że później możecie zmienić zdanie i to bez zagrożenia, że w kodzie napisanym przez inne osoby pojawią się jakieś błędy! Wyobraźcie sobie sytuację, w której połowa osób w firmie używa waszej klasy, której składowe są publiczne, a w pewnego dnia zdacie sobie sprawę, że: „Ups... z tą wartością dzieje się coś, czego nie przewidzieliśmy... chyba musimy zacząć korzystać z metody ustawiającej”. Zapewne kod połowy osób w firmie przestalby przez to działać. W hermetyzacji najlepsze jest to, że później można zmienić zdanie. I nikomu nie przysporzy to żadnych problemów. Zwiększenie efektywności związane z bezpośrednim odwoływaniem się do zmiennych jest nieznaczne i rzadko kiedy — jeśli w ogóle — warto zrezygnować ze stosowania metod ustawiających na jego korzyść.

Hermetyzacja klasy

DobryPiesek

składowe powinny być oznaczone jako prywatne.

metody zwracające i ustawiające powinny być oznaczone jako publiczne.

Choć tak naprawdę metody nie dodają żadnych nowych możliwości funkcjonalnych, to najlepsze w nich jest to, że w przyszłości będziesz mógł zmienić zdanie. Możesz później ponownie zająć się tymi metodami i zmienić je tak, aby były bezpieczniejsze, szybsze i lepsze.

W dowolnym miejscu kodu, w którym została użyta konkretna wartość, można także zastosować metodę zwracającą wartość wynikową tego samego typu.

Zamiast:

int x = 3 + 24;

można napisać:

int x = 3 + pierwszy.getWielkosc();

```
class DobryPiesek {  
  
    private int wielkosc;  
  
    public int getWielkosc() {  
        return wielkosc;  
    }  
  
    public void setWielkosc(int w) {  
        wielkosc = w;  
    }  
}
```

DobryPiesek
wielkosc
getWielkosc() setWielkosc() szczekaj()

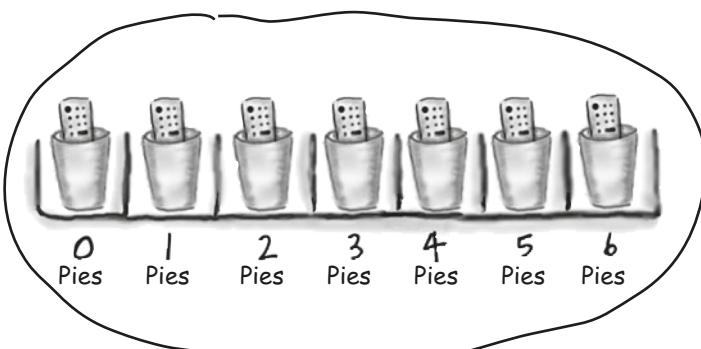
```
void szczekaj() {  
    if (wielkosc > 23) {  
        System.out.println("Houu! Houu!");  
    } else if (wielkosc > 6) {  
        System.out.println("Chau! Chau!");  
    } else {  
        System.out.println("Hiau! Hiau!");  
    }  
}  
  
class DobryPiesekTester {  
    public static void main(String[] args) {  
        DobryPiesek pierwszy = new DobryPiesek();  
        pierwszy.setWielkosc(70);  
        DobryPiesek drugi = new DobryPiesek();  
        drugi.setWielkosc(8);  
        System.out.println("Pierwszy Pies: " + pierwszy.getWielkosc());  
        System.out.println("Drugi Pies: " + drugi.getWielkosc());  
        pierwszy.szzekaj();  
        drugi.szzekaj();  
    }  
}
```

Jak zachowują się obiekty w tablicy?

Tak samo jak wszystkie inne obiekty. Jedyna różnica polega na tym, w jaki sposób można się do nich dostać. Innymi słowy, chodzi o to, w jaki sposób można zdobyć pilot do takich obiektów. Spróbujmy wywołać metodę obiektu Pies umieszczonego w tablicy.

- Deklarujemy i tworzymy tablicę obiektów Pies, zawierającą 7 odwołań do tych obiektów.

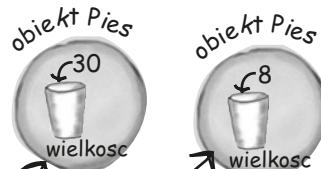
```
Pies[] zwierzaki;  
zwierzaki = new Pies[7];
```



Tablica obiektów typu Pies (Pies[])

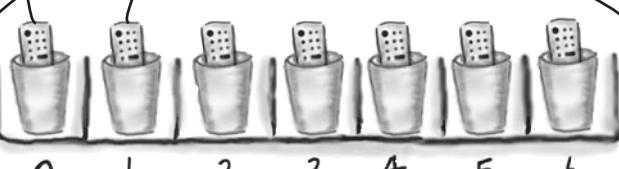
- Tworzymy dwa nowe obiekty Pies i zapisujemy je w dwóch pierwszych elementach tablicy.

```
zwierzaki[0] = new Pies();  
zwierzaki[1] = new Pies();
```



- Wypołujemy metody obu utworzonych wcześniej obiektów Pies.

```
zwierzaki[0].setWielkosc(30);  
int x = zwierzaki[0].getWielkosc();  
zwierzaki[1].setWielkosc(8);
```



Tablica obiektów typu Pies (Pies[])

Deklarowanie i inicjalizacja składowych

Wiesz już, że deklaracja zmiennej wymaga podania, co najmniej, jej nazwy i typu:

```
int wielkosc;  
String imie;
```

Wiesz także, że jednocześnie można zainicjalizować zmienną (czyli przypisać jej pewną wartość):

```
int wielkosc = 420;  
String imie = "Hubert";
```

Jednak co się dzieje w sytuacji, gdy składowa obiektu nie zostanie zainicjalizowana, a wywołamy metodę zwracającą? Innymi słowy, jaka jest wartość składowej, *zanim* zostanie zainicjalizowana?

```
class BiednyPies {  
  
    private int wielkosc; ← Deklaracje dwóch składowych,  
    private String imie;   lecz bez określenia ich  
                          wartości.  
  
    public int getWielkosc() {  
        return wielkosc;  
    }  
  
    public String getImie() {  
        return imie;  
    }  
  
}
```

```
public class BiednyPiesTester {  
    public static void main(String[] args) {  
        BiednyPies psinka = new BiednyPies();  
        System.out.println("Pies ma wielkość: " + psinka.getWielkosc());  
        System.out.println("Pies ma na imię: " + psinka.getImie());  
    }  
}
```

Wiersz polecenia

```
T:\>java BiednyPiesTester  
Pies ma wielkość: 0  
Pies ma na imię: null
```

Składowe zawsze uzyskują wartości domyślne. Zatem nawet jeśli jawnie nie określisz wartości składowej ani nie wywołasz metody ustawiającej, to składowa i tak będzie mieć wartość!

liczby całkowite	0
liczby zmienoprzecinkowe	0.0
składowe logiczne	false
odwołania	null

Jak myślisz? Czy taki kod w ogóle się skompiluje?

Nie musisz inicjalizować składowych, gdyż zawsze mają one wartości domyślne. Składowe liczbowe typów podstawowych (w tym także typu char) przyjmują wartość 0, składowe logiczne (typu boolean) wartość false, a odwołania do obiektów — wartość null. (Pamiętaj, że wartość null oznacza, że pilot nie steruje — nie jest zaprogramowany do obsługi — żadnego obiektu. To odwołanie, ale bez żadnego obiektu docelowego).

Różnica pomiędzy składowymi a zmiennymi lokalnymi

- 1** Składowe są deklarowane wewnątrz klasy a nie wewnątrz metody.

```
class Kon {
    private double wysokosc = 15.2;
    private String rasa;
    // pozostała część kodu
}
```

- 2** Zmienne lokalne są deklarowane wewnątrz metody.

```
class Dodawanie {
    int a;
    int b = 12;

    public int dodaj() {
        int suma = a + b;
        return suma;
    }
}
```

- 3** Przed użyciem **zmiennych lokalnych** koniecznie trzeba je zainicjalizować.

```
class Test {
    public void jazda() {
        int x;
        int z = x + 3;
    }
}
```

To się nie skompiluje!!
Można zadeklarować zmienną x bez przypisywania jej wartości, ale w momencie, gdy spróbujesz jej UŻYĆ, kompilator zgłosi błąd.

```
Wiersz polecenia
F:\>javac Test.java
Test.java:4: variable x might not have been initialized
    int z = x + 3;
           ^
1 error
```

Zmiennym lokalnym nie są przypisywane żadne wartości domyślne. Kompilator zgłosi błąd, jeśli spróbujesz użyć zmiennej lokalnej, zanim zostanie ona zainicjalizowana.

Nie istnieją głupie pytania

P: A co z parametrami metod? W jaki sposób odnoszą się do nich reguły dotyczące zmiennych lokalnych?

O: W zasadzie parametry metod są zmiennymi lokalnymi — są one deklarowane wewnątrz metody (hmm... z technicznego punktu widzenia są one deklarowane nie w kodzie metody, lecz na *liście argumentów*, jednak nie zmienia to faktu, że są one zmiennymi lokalnymi, a nie składowymi). Jednak parametry metod zawsze będą zainicjalizowane; a zatem nigdy nie zobaczysz komunikatu o błędzie informującego, że jakaś zmienna będąca parametrem metody nie została zainicjalizowana.

Jednak wynika to wyłącznie z tego, że kompilator wygeneruje błąd, jeśli spróbujesz wywołać metodę bez przekazania do niej wszystkich niezbędnych argumentów.

A zatem parametry metod ZAWSZE są inicjalizowane, gdyż kompilator wymusza, aby w wywołaniach metod zawsze były podawane wszystkie zadeklarowane argumenty, a wartości argumentów są (automatycznie) przypisywane parametrom.

Porównywanie zmiennych (typów podstawowych oraz odwołań)

Czasami będziesz chciał wiedzieć, czy dwie wartości *typów podstawowych* są sobie równe. Można to sprawdzić w bardzo łatwy sposób — wystarczy użyć operatora `==`. Czasami będziesz chciał sprawdzić, czy dwa odwołania wskazują na ten sam obiekt na stercie. Także to zadanie jest bardzo proste — wystarczy posłużyć się operatorem `==`. Jednak czasami będziesz chciał sprawdzić, czy dwa *obiekty* są takie same, a do tego będzie Ci potrzebna metoda `equals()`. Pojęcie równości obiektów zależy od typu obiektów. Na przykład, jeśli dwa obiekty zawierają te same ciągi znaków (przykładowo „ekspedycja”), to pod względem swego znaczenia i zawartości są takie same, niezależnie od tego, że zajmują zupełnie inne miejsca na stercie. Ale co z naszym obiektem *Pies*? Czy chcemy postrzegać dwa obiekty tego typu jako takie same, jeśli będą mieć taką samą wielkość i wagę? Zapewne nie. A zatem czynniki określające, czy dwa obiekty powinny być uważały za takie same, zależą od ich typu. Sposoby określania równości obiektów poznamy w kolejnych rozdziałach książki (oraz w dodatku B), jak na razie powinieneś jednak zapamiętać, że operator `==` służy *wyłącznie* do porównywania ciągów bitów stanowiących wartości dwóch zmiennych. To, co te ciągi reprezentują, nie ma w tym przypadku znaczenia. Ciagi bitów mogą być takie same albo różne.

Operatora `==` można używać do porównywania dwóch wartości typów podstawowych lub do sprawdzania, czy dwa odwołania wskazują ten sam obiekt.

Do sprawdzenia, czy dwa obiekty są równe, należy używać metody `equals()`.

(Na przykład przy użyciu tej metody można sprawdzić, czy dwa obiekty `String` zawierają ciąg znaków „Flintstone”).

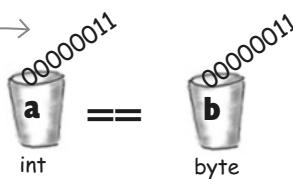
Aby porównać dwie wartości typów podstawowych, należy użyć operatora `==`

Operator `==` można wykorzystać do porównania dwóch zmiennych dowolnych typów, gdyż porównuje on jedynie ciągi bitów.

Warunek w instrukcji `if (a == b) {...}` sprawdza ciągi bitów stanowiące wartości zmiennych a oraz b i zwraca wartość `true`, jeśli są one takie same (operator ten nie zwraca uwagi na wielkości zmiennych, a zatem wszystkie dodatkowe zera znajdujące się z lewej strony wartości nie są uwzględniane).

```
int a = 3;
byte b = 3;
if (a == b) { // to prawda }
```

(Z lewej strony liczby typu int jest więcej zer, jednak w tym przypadku nie zwracamy na nie uwagi)



Aby sprawdzić, czy dwa odwołania są takie same

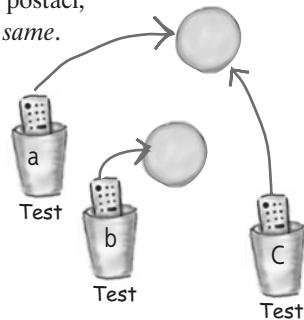
(co oznacza, że wskazują na ten sam obiekt), należy użyć operatora `==`

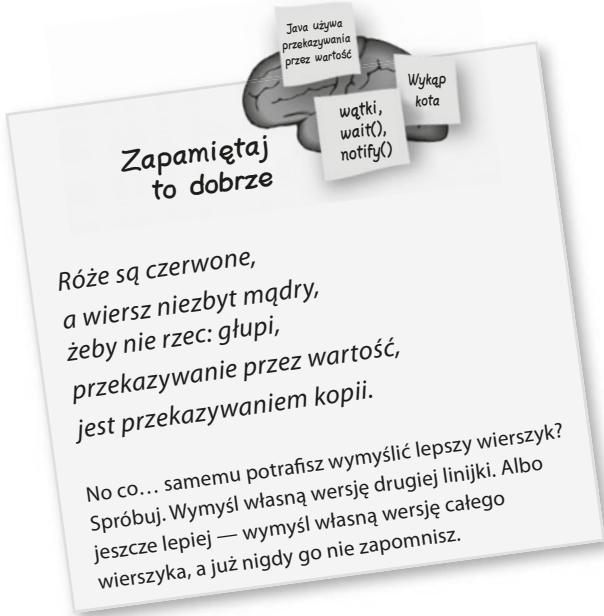
Pamiętaj, że operator `==` zwraca uwagę wyłącznie na ciąg bitów stanowiący wartość zmiennej. Zasada ta pozostaje niezmieniona niezależnie od tego, czy zmienna zawiera odwołanie czy też wartość jednego z typów podstawowych. A zatem operator `==` zwróci wartość `true`, jeśli dwa odwołania wskazują na ten sam obiekt! W takim przypadku nie wiemy, czym jest ciąg bitów (gdź ich postać zależy od JVM i nie jest dostępna), jednak wiemy, że niezależnie od jego postaci, jeśli dwa odwołania wskazują na ten sam obiekt, to reprezentujące je ciągi bitów będą takie same.

```
Test a = new Test();
Test b = new Test();
Test c = a;
if (a == b) { // fałsz }
if (a == c) { // prawda }
if (b == c) { // fałsz }
```

`a == c — to prawda`
`a == b — to falsz`

Ciągi bitów stanowiące zawartość zmiennych a i c są takie same, zatem w przypadku porównania ich przy użyciu operatora `==` okaże się, że są one równe.





Zaostrz ołówek

Co jest poprawne?

Na podstawie podanej poniżej definicji metody określ, które z wywołań podanych z prawej strony są poprawne.

Zaznacz te wywołania, które są poprawne. (Niektóre z podanych instrukcji służą do przypisania wartości zmiennych, które następnie będą użyte w wywołaniach).

```
int obliczPole(int wysokosc, int szerokosc) {  
    return wysokosc * szerokosc;  
}
```



```
int a = obliczPole(7, 12);  
  
short c = 7;  
  
obliczPole(c, 15);  
  
int d = obliczPole(57);  
  
obliczPole(2,3);  
  
long t = 42;  
  
int f = obliczPole(t,17);  
  
int g = obliczPole();  
  
obliczPole();  
  
byte h = obliczPole(4,20);  
  
int j = obliczPole(2,3,5);
```



BĄDŹ kompilatorem



Każdy z plików przedstawionych na tej stronie stanowi niezależny, kompletny plik źródłowy. Twoim zadaniem jest stać się kompilatorem i określić, czy przedstawione programy skompilują się czy nie. Jeśli nie można ich skompilować, to jak je poprawić? Jeśli można je skompilować, to jakie wygenerują wyniki?

A

```
class XCopy {  
    public static void main(String[] args) {  
        int org = 42;  
        XCopy x = new XCopy();  
        int y = x.jazda(org);  
        System.out.println(org + " " + y);  
    }  
  
    int jazda(int arg) {  
        arg = arg * 2;  
        return arg;  
    }  
}
```

B

```
class Zegar {  
    String czas;  
  
    void setCzas(String c) {  
        czas = c;  
    }  
  
    void getCzas() {  
        return czas;  
    }  
}  
  
class ZegarTester {  
    public static void main(String[] args) {  
  
        Zegar z = new Zegar();  
  
        z.setCzas("1245");  
        String dta = z.getCzas();  
        System.out.println("Czas: " + dta);  
    }  
}
```



Kim jestem?



Grupa zamaskowanych komponentów Javy gra w grę towarzyską o nazwie „Zgadnij, kim jestem”. Komponenty dają podpowiedzi, a Ty na ich podstawie starasz się odgadnąć, kim one są. Załóż, że komponenty zawsze mówią prawdę. Jeśli mówią coś, co może być prawdą w odniesieniu do kilku z nich, wybierz wszystkie komponenty, dla których podane stwierdzenie jest prawdziwe. W pustych miejscach obok podanych podpowiedzi podaj nazwy komponentów biorących udział w zabawie.

W dzisiejszej zabawie udział biorą:

składowa, argument, wartość wynikowa, metoda ustawiająca, metoda zwracająca, hermetyzacja, modyfikator public, modyfikator private, przekazywanie przez wartość, metoda.

Klasa może mieć ich dowolną ilość.

Metoda może mieć tylko jedną z nich.

Mogą być niejawnie rozszerzane.

Wolę, aby moje składowe były prywatne.

W rzeczywistości oznacza „utworzenie kopii”.

Jedynie metody ustawiające powinny je aktualizować.

Metody mogą mieć ich dowolną ilość.

Z definicji coś zwracam.

Nie należy mnie stosować wraz ze składowymi.

Mogę mieć wiele argumentów.

Z definicji wymagam podania tylko jednego argumentu.

Pomagają w hermetyzacji.

Ja zawsze działam sam.



Pomieszane komunikaty

Obok zamieszczono prosty program w Javie. Brakuje w nim dwóch fragmentów. Twoim zadaniem jest dopasowanie proponowanych bloków kodu (przedstawionych poniżej) z wynikami, które program wygeneruje po wstawieniu wybranego bloku.

Nie wszystkie wiersze wyników zostaną wykorzystane, a niektóre z nich mogą być wykorzystane więcej niż jeden raz. Narysuj linie łączące bloki kodu z odpowiadającymi im wynikami.

Proponowane fragmenty kodu

x < 9

indeks < 5

x < 20

indeks < 5

x < 7

indeks < 7

x < 19

indeks < 1

Możliwe dane wynikowe

14 7

9 5

19 1

14 1

25 1

7 7

20 1

20 5

```
public class Mix4 {  
    int licznik = 0;  
  
    public static void main(String[] args) {  
        int ilosc = 0;  
        Mix4[] m4a = new Mix4[20];  
        int x = 0;  
        while ( [ ] ) {  
            m4a[x] = new Mix4();  
            m4a[x].licznik = m4a[x].licznik + 1;  
            ilosc = ilosc + 1;  
            ilosc = ilosc + m4a[x].mozeNowa(x);  
            x = x + 1;  
        }  
        System.out.println(ilosc + " "  
                           + m4a[1].licznik);  
    }  
  
    public int mozeNowa(int indeks) {  
        if ( [ ] ) {  
            Mix4 m4 = new Mix4();  
            m4.licznik = m4.licznik + 1;  
            return 1;  
        }  
        return 0;  
    }  
}
```



Zagadkowy basen



Twoim **zadaniem** jest wybranie fragmentów kodu z basenu i umieszczenie ich w miejscach kodu oznaczonych podkreśleniami. żadnego fragmentu kodu **nie można** użyć więcej niż raz, a co więcej, nie wszystkie fragmenty zostaną wykorzystane. **Zadanie** polega na stworzeniu klasy, którą będzie można skompilować i która wygeneruje wyniki przedstawione poniżej.

Wyniki:

```
cmd Wiersz polecenia
T:\>java ZagadkowyBasen4
Wynik 543345
```

Notatka: Każdy fragment kodu z basenu może zostać użyty tylko raz!

```
class ZagadkowyBasen4 {
    public static void main(String[] args) {
```

```
        int y = 1;
        int x = 0;
        int wynik = 0;
        while (x < 6) {
```

```
        _____
```

```
    }
    x = 6;
    while (x > 0) {
```

```
        wynik = wynik + _____
```

```
    }
    System.out.println("Wynik " + wynik);
}
```

```
class _____ {
```

```
    int izm;
```

```
    _____ zrobCos(int _____) {
```

```
        if (izm > 100) {
```

```
            return _____
```

```
        } else {
```

```
            return _____
```

```
}
```

```
}
```

```
zrobCos(x);
obty.zrobCos(x);
obty[x].zrobCos(czynnik);
obty[x].zrobCos(x); x = x + 1;
izm + czynnik; izm = x; x = x - 1;
izm * (2 + czynnik); obty.izm = x; ZagadkowyBasen4 izm
izm * (5 - czynnik); obty[x].izm = x; ZagadkowyBasen4b czynnik
izm * czynnik; obty[x].izm = y; ZagadkowyBasen4b() public int
ZagadkowyBasen4[] obty = new ZagadkowyBasen4[6]; ZagadkowyBasen4b() private short
ZagadkowyBasen4b[] obty = new ZagadkowyBasen4b[6];
ZagadkowyBasen4b[] obty = new ZagadkowyBasen4[6];
obty[x] = new ZagadkowyBasen4b(x);
obty[] = new ZagadkowyBasen4b();
obty[x] = new ZagadkowyBasen4b();
obty = new ZagadkowyBasen4b();
```





Trudne czasy w Używkowie

Gdy Kukuła dźgnął go swoim coltem w bok, Jaś zamarł w bezruchu. Wiedział, że Kukuła jest równie głupi co paskudny i nie chciał drażnić osiąka. Kukuła kazał Jasiowi iść do biura swojego szefa, jednak Jaś nie zrobił niczego złego (przynajmniej ostatnio), doszedł zatem do wniosku, że krótka pogawędka z szefem Kukuly — Chudzizną — niczemu nie powinna zaszkodzić. Jaś rozprowadzał ostatnio wiele stymulatorów neuronowych w zachodniej dzielnicy i mógł oczekiwany, że Chudzizna powinien być z tego zadowolony. Handel stymulatorami z czarnego rynku nie był co prawda najbardziej dochodowym interesem, jaki można sobie wyobrazić, jednak był raczej bezpieczny. Większość ludzi biorących te środki, których Jaś widział, szybko wracała od pełnej świadomości, choć może byli nieco mniej skoncentrowani niż wcześniej.

„Biuro” Chudzizny było śmierdzącą norą, jednak kiedy Jaś został do niego wepnieniety, zauważył, że zostało zmodyfikowane i wyposażone we wszystkie bajery, jakich mógł potrzebować lokalny „boss”. — Jasiu, mój chłopcze — syknął Chudzizna — miło Cię znowu widzieć. — Mnie również, jak sądzę...

— odpowiedziała Jaś, wyczuwając złośliwość w głosie Chudzizny. — Panie Chudzizna, jesteśmy chyba kwita, czy o czymś zapomniałem? — Ej! W końcu dobrze wyglądasz, przytyleś trochę... Ja, niestety, stałem się ostatnio ofiarą, jakby to nazwać, niewielkiego naruszenia prywatności... — powiedział Chudzizna.

Jaś mimowolnie drgnął, swego czasu był w końcu jednym z najlepszych hakerów.

Za każdym razem, kiedy komuś udało się dokonać jakiegoś komputerowego „włamu”, niepożądana uwaga kierowała się właśnie na Jasia. — To na pewno nie byłem ja — powiedział Jaś. — Ta zabawa jest zbyt ryzykowna. Już się wycofałem z tego interesu. Mam swoje zajęcie i nie wtykam nosa w nie swoje sprawy. — Ha, ha... — zaśmiał się Chudzizna. — Nie wątpię, że w tej sprawie jesteś czysty, ale będę miał poważne straty, dopóki ten haker nie zostanie sprzątnięty! — Cóż, zyczę powodzenia, panie Chudzizna, ale może jednak powiniem mnie pan podrzucić do domu, żeby mogłem opchnąć kilka działek państwowego towaru, póki jeszcze będę dziś w stanie... — powiedział Jaś. — Obawiam się Jasiu, że to nie takie proste. Kukuła mówił, że podobno bawiłeś się J37WN — zasugerował Chudzizna.

— Wersja neuronowa? Trochę się nią zajmowałem. I co z tego? — odpowiedział Jaś, czując lekki ucisk w żołądku. — Za pomocą wersji neuronowej daję klientom znać, gdzie będzie następna dostawa — wyjaśnił Chudzizna. — Problem polega na tym, że niektórzy z tych cwaniaczków biorących stymulatory neuronowe są podłączeni wystarczająco długo, aby wykombinować, jak włamać się do mojej bazy danych. Potrzebuję takiego bystrzaka jak ty, Jasiu, żeby zerknął na moją klasę DostawaJ37WN — metody, składowe i cały ten kram, i żeby sprawdził, jak ci goście się dobierają do moich danych. Trzeba...

— Hej! — wrzasnął Kukuła — nie chcę, żeby jakiś jajogłowy harcerzyk jak Jasio grzebał w moim kodzie! — Spokojnie, stary — Jasiu doszedł do wniosku, że teraz nadszedł czas na zabranie głosu.

— Na pewno zrobiłeś, co trzeba ze swoimi modyfi... — Nie mów mi, co mam robić! — jeszcze głośniej krzyknął Kukuła. — Wszystkie metody dla tych ćpunów zostawiłem jako publiczne, żeby mieli dostęp do informacji o miejscu dostawy, ale wszystkie metody operujące na bazie są prywatne. Nikt z zewnątrz nie może się dobrać do tych metod. Nikt!

— Myślę, Chudzizna, że mogę Ci pomóc. Co ty na to, żebyśmy zostawili Kukułę tu na rogu, a sami przeszliśmy na mały spacerek? — zasugerował Jasiu. Kukuła siegnął po swojego colta, ale Chudzizna już trzymała broń przy jego skroni. — Odlóż spluwę, Kukuła — syknął. — Jasio i ja mamy kilka spraw do obgadania.

Co podejrzewała Jasiu?

Czy wyjdzie z meliny Chudzizny w „jednym kawałku”?



Ćwiczenie rozwiązanie

A

Klasa XCopy kompliuje się i działa poprawnie w podanej postaci. Generowane wyniki to „42 84”. Pamiętaj, że w Javie argumenty są przekazywane przez wartość (co oznacza, że przekazywana jest kopia wartości); zmienna org nie jest zmieniana przez wywołanie metody jazda().

```
class Zegar {  
    String czas;  
  
    void setCzas(String c) {  
        czas = c;  
    }  
  
    String getCzas() {  
        return czas;  
    }  
}
```

```
class ZegarTester {  
    public static void main(String[] args) {  
        Zegar z = new Zegar();  
  
        z.setCzas("1245");  
        String dta = z.getCzas();  
        System.out.println("Czas: " + dta);  
    }  
}
```

**Uwaga: Metoda zwracająca z założenia
musi mieć jakąś wartość wynikową.**

Klasa może mieć ich dowolną ilość

**składowa, metoda ustawiająca, metoda zwracająca, metoda
wartość wynikowa**

Metoda może mieć tylko jedną z nich

wartość wynikowa, argument

Mogą być niejawnie rozszerzane

hermetyzacja

Wolę, aby moje składowe były prywatne

przekazywanie przez wartość

W rzeczywistości oznacza „utworzenie kopii”

składowa

Jedynie metody ustawiające powinny je aktualizować

argument

Metody mogą mieć ich dowolną ilość

metoda zwracająca

Z definicji coś zwracam

modyfikator public

Nie należy mnie stosować wraz ze składowymi

metoda

Mogę mieć wiele argumentów

metoda ustawiająca

Z definicji wymagam podania tylko jednego argumentu

**metoda zwracająca, metoda ustawiająca, modyfikator
public, modyfikator private**

Pomagają w hermetyzacji

wartość wynikowa

Ja zawsze działam sam



Rozwiążanie zagadki

```

class ZagadkowyBasen4 {
    public static void main(String[] args) {
        ZagadkowyBasen4b[] obty = new ZagadkowyBasen4b[6];
        int y = 1;
        int x = 0;
        int wynik = 0;
        while (x < 6) {
            obty[x] = new ZagadkowyBasen4b();
            obty[x].izm = y;
            y = y * 10;
            x = x + 1;
        }
        x = 6;
        while (x > 0) {
            x = x - 1;
            wynik = wynik + obty[x].zrobCos(x);
        }
        System.out.println("Wynik " + wynik);
    }
}

class ZagadkowyBasen4b {
    int izm;
    public int zrobCos(int czynnik) {
        if (izm > 100) {
            return izm * czynnik;
        } else {
            return izm * (5 - czynnik);
        }
    }
}

```

Wiersz polecenia

```
T:\>java ZagadkowyBasen4
Wynik 543345
```

Zagadka na pięć minut. Rozwiążanie

Jasiu wiedział, że Kukuła nie jest najbystrzejszym koderem. Kiedy słuchał, co Kukuła mówił o swoim kodzie, zauważył, że nie wspominał nic o składowych. Podejrzewał, że choć Kukuła faktycznie poprawnie zdefiniował metody, to jednak zapomniał oznaczyć składowych jako prywatne. Ten błąd mógł bardzo drogo kosztować Chudziznę.

Proponowane fragmenty kodu

x < 9

indeks < 5

x < 20

indeks < 5

x < 7

indeks < 7

x < 19

indeks < 1

Możliwe dane wynikowe

14 7

9 5

19 1

14 1

25 1

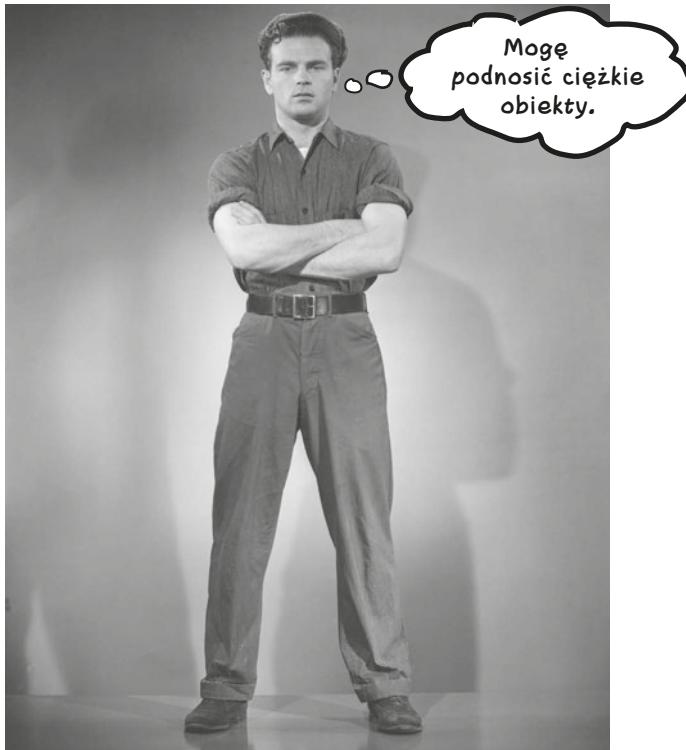
7 7

20 1

20 5

5. Pisanie programu

Supermocne metody



Dodajmy naszym metodom nieco siły. Igraliśmy ze zmiennymi, zabawialiśmy się z kilkoma obiektami i napisaliśmy parę wierszy kodu. Ale potrzeba nam więcej narzędzi. Takich jak **operatory**. Operatory są nam potrzebne, abyśmy mogli pisać coś ciekawszego niż metody takie jak szczekaj(). Oraz **pętle**. Potrzebujemy pętli, ale co się dzieje z tymi niepozornymi pętlami *while*? Jeśli jesteśmy naprawdę poważni, to będziemy także potrzebować pętli **for**. Może się nam przydać umiejętność **generowania liczb losowych**. A co z **zamienianiem łańcuchów znaków na liczby**? O... to też by było super. Tego też trzeba by się nauczyć. A czy nie można by się nauczyć tego wszystkiego, pisząc jakiś program? Tak żeby zobaczyć, jak wygląda pisanie normalnego programu od samego początku. **Na przykład jakąś grę**... taką jak gra w okręty. Napisanie gry to problem na dużą skalę, dlatego jego rozwiążanie zajmie nam aż dwa rozdziały. W tym stworzymy wersję uproszczoną, a w dalszej części książki — w rozdziale 6. — stworzymy wersję pełną o znacznie większych możliwościach.

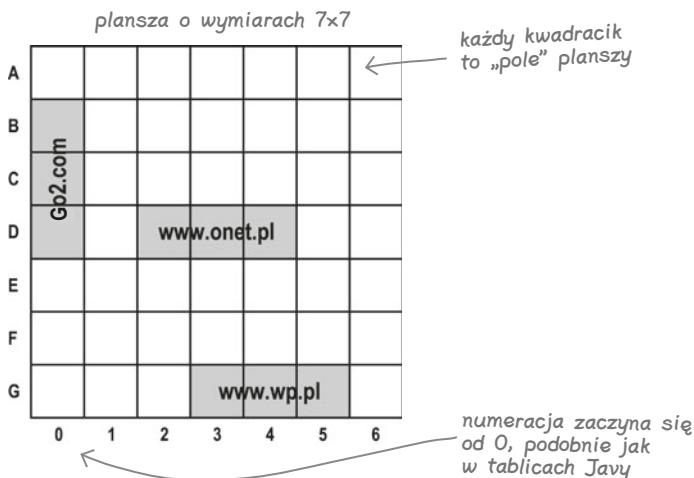
Napiszmy grę przypominającą „statki”, o nazwie „Zatopić portal”

W grze zmagasz się z komputerem, ale w odróżnieniu od normalnej gry w statki, w tej nie umieszczasz na planszy własnych okrętów. Twoim zadaniem jest natomiast zatopienie okrętów komputera w jak najmniejszej ilości ruchów. Poza tym nie zatapiamy statków. Zatapiamy „portale” (czyli duże witryny internetowe).

Cel: Zatopić wszystkie portale komputera w jak najmniejszej ilości ruchów. Gra wyznacza ocenę gracza na podstawie uzyskanych przez niego wyników.

Przygotowanie do gry: W momencie uruchamiania gry komputer umieszcza na **wirtualnej planszy o wymiarach 7×7** trzy portale. Kiedy proces ten zostanie zakończony, gracz może już podać pierwsze pole, w jakie celuje.

Zasady gry: Jeszcze nie nauczyliśmy się tworzyć graficznego interfejsu użytkownika w naszych programach, zatem ta wersja programu jest obsługiwana z poziomu wiersza polecień. Komputer poprosi Cię o wskazanie „ostrzeliwanego” pola planszy, a Ty w odpowiedzi wpisujesz jego współrzędne, na przykład: A3 lub D5 i tak dalej. W odpowiedzi na podane pole na ekranie zostaną wyświetlane rezultaty — „trafiony”, „pułdo” lub „zatopiłeś onet.pl” (przy czym podana nazwa może być inna, zależnie od tego, jaki portal tym razem miał pecha). Kiedy wszystkie trzy portale zostaną zatopione i zostanie po nich jedynie rozwiewany wiatrem obłoczek w kształcie liczby 404, gra zakończy się, a program wyświetli ocenę gracza.



Masz za zadanie napisać grę „Zatopić portal”, przy czym każdy z zatapianych portali zajmuje dokładnie trzy pola planszy.

Fragment gry:

```
I:\>java PortalGraMax
Twoim celem jest zatopienie trzech portali.
onet.pl Goo2.com wp.com
Postaraj się je zatopić w jak najmniejszej ilości ruchów.
Podaj pole: d1
pułdo
Podaj pole: e2
pułdo
Podaj pole: f0
trafiony
Podaj pole: f1
trafiony
Podaj pole: f2
pułdo
Huu! Zatopiłeś portal onet.pl :(
zatopiony
Podaj pole: h4
pułdo
Podaj pole: c3
trafiony
Podaj pole: b3
trafiony
Podaj pole: a3
Huu! Zatopiłeś portal wp.com :(
zatopiony
```

Na początku stworzymy ogólny projekt gry

Wiemy, że będą nam potrzebne klasy i metody, ale jakie? Aby odpowiedzieć na to pytanie, będziemy potrzebowali więcej informacji na temat tego, jak nasza gra ma działać. W pierwszej kolejności musimy określić ogólny schemat przepływu sterowania w grze. Oto jego podstawowy zarys:

1 Użytkownik rozpoczyna grę.

- A** Program gry tworzy trzy portale.
- B** Program umieszcza portale na wirtualnej planszy.

2 Zaczyna się rozgrywka.

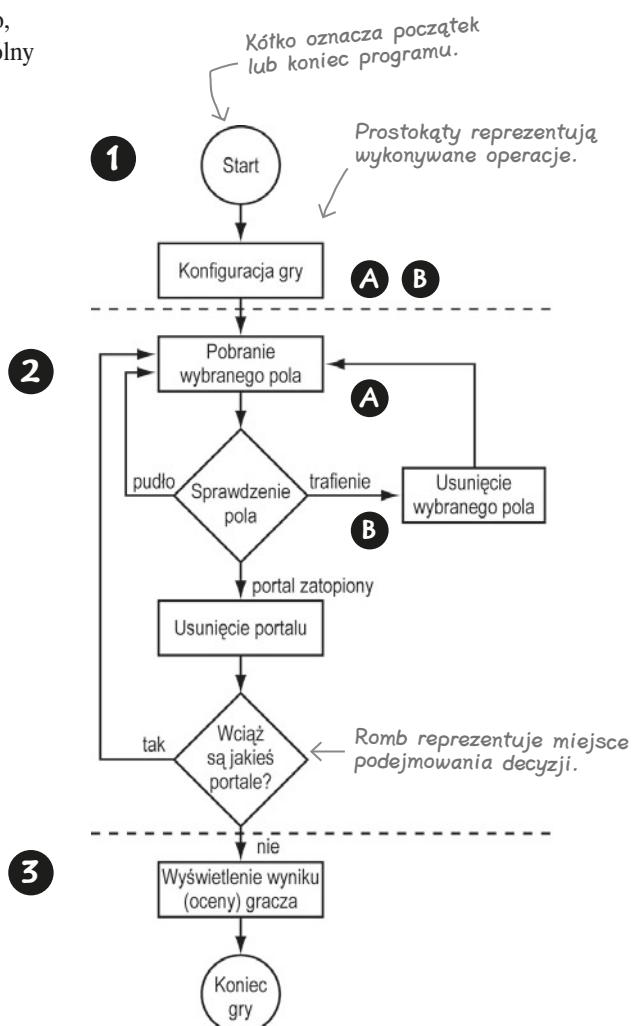
Powtarzaj poniższe czynności tak długo, aż nie będzie już żadnych portali.

- A** Program prosi użytkownika o wskazanie pola (na przykład: A2, C0 itd.).
- B** Program porównuje pole wskazane przez użytkownika z położeniem wszystkich trzech portali i sprawdza, czy użytkownik chybił czy trafił. Następnie wykonuje odpowiednie czynności — jeśli portal został trafiony, to wskazane pole (A2, D4 itd.) jest usuwane, jeśli wszystkie pola zajmowane przez portal zostały trafione, usuwany jest cały portal.

3 Koniec gry.

Program wyświetla ocenę gracza określona na podstawie ilości ruchów.

Teraz już się orientujesz, jakie operacje program musi wykonywać. Kolejnym zadaniem jest określenie, jakie **obiekty** będą nam potrzebne do wykonania tych czynności. Pamiętaj, myśl jak Jurek „obiektowiec”, a nie jak Bronek — czyli w pierwszej kolejności myśl o **rzedach** występujących w programie, a nie o **procedurach**.



Łagodne wprowadzenie do prostszej wersji gry

Wydaje na to, że będą nam potrzebne przynajmniej dwie klasy — `PortalGra` oraz `Portal`. Zanim jednak stworzymy grę w jej ostatecznej postaci, zaczniemy od wersji uproszczonej, o nazwie **Uproszczone zatapianie portali**. W tym rozdziale stworzymy prostą wersję gry, natomiast wersja pełna zostanie przedstawiona w rozdziale *następnym*.

W tej wersji gry wszystko jest prostsze. Zamiast dwuwymiarowej planszy, nazwa portalu jest ukrywana w pojedynczym *wierszu*. A zamiast *trzech* portalu, mamy do zatopienia tylko *jeden*.

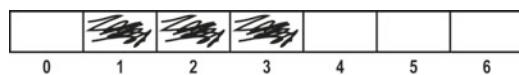
Cel gry pozostaje jednak taki sam. Wciąż zatem musimy stworzyć obiekt `Portal`, określić jego położenie w wierszu, pobrać dane wejściowe od użytkownika, a po trafieniu wszystkich pól zajmowanych przez nazwę portalu — zakończyć grę. Ta uproszczona wersja programu stanowi doskonały punkt startowy do rozpoczęcia pisania pełnej wersji gry. Jeśli uda nam się ją uruchomić, to w przyszłości będziemy także w stanie ją rozbudować.

W tej uproszczonej wersji gry główna klasa gry nie ma żadnych składowych, a cały kod obsługujący grę został umieszczony w metodzie `main()`. Innymi słowy, po uruchomieniu programu i rozpoczęciu wykonywania metody `main()` program utworzy jedną, i tylko jedną, kopię obiektu `Portal`, określi jej położenie (trzy sąsiadujące ze sobą komórki wirtualnego wiersza składającego się z siedmiu komórek), zapyta użytkownika o pole, jakie należy sprawdzić, skontroluje, czy cel został trafiony, i będzie powtarzał ostatnie czynności aż do momentu, gdy wszystkie trzy komórki zostaną trafione.

Pamiętaj, że wirtualny wiersz jest... *wirtualny*. Innymi słowy, nie istnieje on w programie. O ile tylko zarówno program, jak i gra „wiedzą”, że `Portal` został ukryty w trzech sąsiednich komórkach wiersza (spośród siedmiu, przy czym pierwsza z nich ma indeks zero), to sam wiersz nie musi być reprezentowany w kodzie programu. Możesz czuć pokusę, aby stworzyć tablicę siedmiu liczb całkowitych, a następnie umieścić `Portal` w trzech spośród nich, jednak nie jest to konieczne. W rzeczywistości potrzeba nam jedynie tablicy zawierającej trzy komórki, które zajmuje `Portal`.

- 1 **Gra się rozpoczyna** i tworzony jest jeden obiekt `ProstyPortal`, który zostaje umieszczony w trzech spośród siedmiu komórek wiersza.

Zamiast współrzędnych o postaci „A2” lub „C4” i tak dalej położenie jest określane przy użyciu tylko jednej liczby całkowitej (na przykład, w poniższym przykładzie nasz cel zajmuje komórki 1, 2 i 3).



- 2 **Rozpoczyna się gra**. Program prosi użytkownika o podanie komórki, a następnie sprawdza, czy została trafiona którakolwiek z komórek zajmowanych przez `Portal`. Jeśli jedna z tych komórek została trafiona, inkrementowana jest wartość zmiennej `iloszTrafien`.

- 3 **Gra się kończy**, kiedy wszystkie trzy komórki zostaną trafione (czyli gdy zmienna `iloszTrafien` przyjmie wartość 3). W takim przypadku na ekranie wyświetlana jest informacja o tym, w ilu ruchach użytkownikowi udało się zatopić `Portal`.

Pełny przykład jednej rozgrywki.

```
ca. Wiersz polecenia
T:>java ProstyPortalGra
Podaj liczbę 3
pułko
Podaj liczbę 4
trafiony
Podaj liczbę 5
trafiony
Podaj liczbę 6
zatopiony
4 ruchów
```

Tworzenie klasy

Jako programista masz zapewne jakąś metodologię (proces lub sposób) pisania kodu. Także my mamy taką metodologię. Nasza sekwencja czynności została zaprojektowana w taki sposób, abyś mógł łatwiej dowiedzieć się (i nauczyć), o czym myśleliśmy, tworząc klasę. Nie oznacza to wcale, że w taki sam sposób tworzymy (lub *Ty* tworzysz) klasy, pisząc prawdziwe programy. Zazwyczaj pisząc normalne programy, będziesz się zapewne kierować własnymi upodobaniami lub wymogami, jakie narzuca tworzony projekt lub pracodawca. Jednak my możemy postąpić całkowicie dowolnie — w sposób, jaki nam najbardziej odpowiada. A kiedy tworzymy klasę Javy, traktując ją w kategoriach „doświadczenia poznawczego”, robimy to zazwyczaj w następujący sposób:

- Określamy, co klasa powinna robić.
- Tworzymy listę **składowych i metod**.
- Piszemy **kod przygotowawczy** metod.
(Już niedługo zobaczysz, jak on wygląda).
- Piszemy **kod testowy** metod.
- Implementujemy** klasę.
- Testujemy** klasę.
- W razie konieczności **poprawiamy błędy i ponownie implementujemy** klasę.
- Wyrażamy głęboką wdzięczność i zadowolenie, że nie musimy zdobywać doświadczenia kosztem żywych użytkowników.



Uaktywnij swoje dendryty.

W jaki sposób, pisząc program, podejmiesz decyzję, jaką klasę (lub klasy) stworzysz w pierwszej kolejności? Zakładając, że wszystkie programy, za wyjątkiem tych najmniejszych, wymagają więcej niż jednej klasy (jak się dzieje, kiedy postępujesz zgodnie z dobrymi zasadami programowania obiektowego i nie tworzysz klas, które służą do różnych celów), to od czego zaczynasz ich tworzenie?

Trzy wersje kodu, jakie stworzymy dla każdej z klas:

kod przygotowawczy kod testowy kod właściwy

Powyższy rysunek jest prezentowany na kolejnych stronach, abyś łatwo mógł zorientować się, nad czym aktualnie pracujesz. Na przykład, jeśli u góry strony zobaczysz następujący rysunek, będziesz wiedzieć, że pracujesz nad kodem przygotowawczym klasy ProstyPortal.



Kod przygotowawczy

To pewien rodzaj pseudokodu, który ma za zadanie pozwolić Ci skoncentrować się nad logiką działania kodu, a nie nad jego składnią.

Kod testowy

To klasy lub metody, które będą służyły do testowania faktycznego kodu i potwierdzą, że działa on poprawnie.

Kod właściwy

Faktyczna implementacja klasy. To kod programu napisany w Javie.

Do zrobienia

Klasa ProstyPortal

- napisz kod przygotowawczy,
- napisz kod testowy,
- napisz ostateczny kod w Javie.

Klasa ProstyPortalGra

- napisz kod przygotowawczy,
- napisz kod testowy [nie],
- napisz ostateczny kod w Javie.

kod przygotowawczy

kod testowy

kod właściwy

ProstyPortal
int [] polaPolozenia int iloscTrafien
String sprawdz(String pole) setPolaPolozenia(int [] ppol)

Czytając poniższy przykład, zrozumiesz, czym jest i jak działa kod przygotowawczy (będący naszą wersją pseudokodu). Jak zobaczyś, jest to coś pośredniego pomiędzy kodem napisanym w Javie a zwyczajnym — słownym — opisem klasy. W większości przypadków kod przygotowawczy składa się z trzech części: deklaracji składowych, deklaracji metod oraz logiki działania metod. Najważniejszą częścią kodu przygotowawczego jest logika działania metod, gdyż to ona określa, co ma się działać, i to właśnie ona zostanie następnie — podczas pisania kodu klasy — przetłumaczona na to, w jaki sposób należy wykonać zaplanowane operacje.

ZADEKLARUJ tablicę typu int przechowującą położenia komórek. Nazwij ją polaPolozenia.

ZADEKLARUJ zmienną typu int przechowującą ilość trafień. Nazwij ją iloscTrafien, **PRZYPISZ** jej wartość 0.

ZADEKLARUJ metodę sprawdz() wymagającą podania łańcucha znaków (obiektu String) określającego pole wiersza wskazane przez użytkownika (na przykład „1” lub „3” itd.). Metoda sprawdza wskazane pole i zwraca wynik reprezentujący „trafienie”, „pułóż” lub „zatopienie”.

ZADEKLARUJ metodę ustawiającą setPolaPolozenia() wymagającą podania tablicy typu int (posiadającej trzy komórki zawierające liczby całkowite typu int, na przykład: 1, 2, 3).

METODA String sprawdz(String pole)

POBIERZ pole wybrane przez gracza jako parametr typu String

SKONWERTUJ pole podane przez użytkownika na liczbę typu int

POWTÓRZ dla każdej z komórek typu int

// **PORÓWNAJ** pole wskazane przez gracza z polami położenia

JEŚLI użytkownik trafił

INKREMENTUJ ilość trafień

// **SPRAWDŹ**, czy to była ostatnia komórka Portal

JEŚLI ilość trafień jest równa 3, **ZWRÓĆ** wynik „zatopiony”

W PRZECIWNYM RAZIE, jeśli Portal nie został zatopiony, **ZWRÓĆ** wynik „trafiony”

KONIEC JEŚLI

W PRZECIWNYM RAZIE pole wskazane przez użytkownika nie jest zajmowane przez Portal, **ZWRÓĆ** wynik „pułóż”

KONIEC JEŚLI

KONIEC POWTÓRZ

KONIEC METODY

METODA: setPolaPolozenia(int[] ppol)

POBIERZ położenia pól przekazane jako parametr (tablica liczb typu int)

PRZYPISZ parametr określający pola położenia do odpowiedniej składowej

KONIEC METODY

kod przygotowawczy

kod testowy

kod właściwy

Pisanie implementacji metod

Napiszmy w końcu faktyczny kod metody i uruchommy to cudeńko.

Zanim zaczniemy pisać kod metod, wstrzymajmy się chwilę i napiszmy kod służący do *testowania* metod. Właśnie tak, mamy zamiar napisać kod testujący, jeszcze *zanim* w ogóle będzie co testować!

Pomyśl pisania kodu testującego w pierwszej kolejności jest jedną z praktyk zalecanych przez programowanie ekstremalne (w skrócie XP, od wyróżnionych liter angielskich słów: *eXtreme Programming*) i może ułatwić (oraz przyspieszyć) tworzenie kodu. Nie twierdzimy, że koniecznie musisz stosować się do zasad programowania ekstremalnego, jednak nam podoba się pomysł pisania kodu testowego w pierwszej kolejności. A w samej nazwie „programowanie ekstremalne” jest coś fajnego.



Programowanie ekstremalne (XP)

Programowanie ekstremalne to nowy przybysz do świata metodologii programistycznych. Uważa się, że zasady programowania ekstremalnego wyznaczają sposób, w jaki „programiści naprawdę chcą pracować”. Metodologia ta pojawiła się pod koniec lat 90-tych i została zastosowana przez wiele firm, zaczynając od sklepów mieszczących się w garażu i prowadzonych przez 2 osoby, a kończąc na Ford Motor Company. Zakłada się, że dzięki niej klienci dostaną to, co chcieli, w czasie, w jakim chcieli, i to nawet, jeśli na końcowych etapach prac zmieni się specyfikacja.

Programowanie ekstremalne bazuje na grupie dobrze sprawdzonych praktyk zaprojektowanych w taki sposób, aby doskonale ze sobą współpracowały. Jednak wiele osób wybiera i wykorzystuje jedynie niektóre spośród nich. Poniżej przedstawiliśmy niektóre spośród praktyk zalecanych przez programowanie ekstremalne:

Często udostępniaj nowe wersje oprogramowania, w których jest wprowadzanych stosunkowo niewiele zmian.

Twórz program w powtarzających się cyklach.

Nie wprowadzaj do programu żadnych możliwości, których nie ma w specyfikacji (niezależnie od tego, jak wielka jest pokusa umieszczenia dodatkowych możliwości funkcjonalnych „na przyszłość”).

Najpierw pisz kod testujący.

Nie narzucaj zabójczego tempa prac, pracuj w normalnych godzinach.

Poprawiaj kod zawsze, gdy tylko zauważysz taką możliwość.

Nie publikuj niczego, dopóki kod nie przechodzi wszystkich testów.

Określaj realistyczny czas prac bazujący na kolejnych wersjach, w których będą wprowadzane niewielkie ilości modyfikacji.

Staraj się, aby pisany kod był jak najprostszy.

Pisz programy w parach i zmieniaj skład poszczególnych par, tak aby, w miarę możliwości, wszyscy wiedzieli wszystko o tworzonym kodzie.

Pisanie kodu testowego dla klasy ProstyPortal

Musimy napisać kod, który będzie w stanie utworzyć obiekt ProstyPortal i wywołać jego metody. W przypadku tej klasy zależy nam jedynie na metodzie sprawdz(), choć aby metoda ta mogła działać poprawnie, będziemy także musieli zaimplementować metodę setPolaPołożenia().

Uważnie przeanalizuj przedstawiony poniżej kod przygotowawczy metody sprawdz() (metoda setPolaPołożenia() jest bardzo prostą metodą ustawiającą, zatem nie będziemy przejmować się jej testowaniem; jednak możliwe, że w „prawdziwej” aplikacji będziemy chcieli stworzyć solidniejszą metodę ustawiającą, którą warto by było przetestować).

Następnie zadaj sobie pytanie: „Gdyby metoda sprawdz() była już zaimplementowana, to jaki powinien być kod testujący, który mógłby potwierdzić poprawność jej działania?”.

Opierając się na poniższym kodzie przygotowawczym:

```
METODA String sprawdz(String podanePole)
POBIERZ pole wskazane przez gracza jako argument typu String
SKONWERTUJ wskazane pole na liczbę typu int
POWTARZAJ dla każdej komórki tablicy typu int określającej położenie Portalu
    // PORÓWNAJ pole wskazane przez gracza z tablicą położenia Portalu
    JEŚLI wskazane pole odpowiada jednej z komórek położenia Portalu
        INKREMENTUJ ilość trafień
        // SPRAWDŹ, czy to była ostatnia komórka położenia Portalu
        JEŚLI ilość trafień wynosi 3, ZWRÓĆ wynik „zatopiony”
        W PRZECIWNYM RAZIE Portal nie został zatopiony, ZWRÓĆ wynik „trafiony”
        KONIEC JEŚLI
    W PRZECIWNYM RAZIE gracz nie trafił, zatem ZWRÓĆ wynik „pudło”
    KONIEC JEŚLI
KONIEC POWTARZAJ
KONIEC METODY
```

Oto, co powinniśmy testować:

1. Stworzenie obiektu klasy ProstyPortal.
2. Określenie położenia obiektu (czyli stworzenie tablicy trzech liczb całkowitych, na przykład {2,3,4}).
3. Stworzenie łańcucha znaków reprezentującego pole wskazane przez gracza (na przykład: „2” lub „0” itp.).
4. Wywołanie metody sprawdz() wraz z przekazaniem do niej pola wskazanego przez gracza.
5. Wyświetlenie wyników w celu sprawdzenia, czy Portal został trafiony (na przykład „zakończony pomyślnie” lub „niepowodzenie”).

Nie istnieja grupie pytania

P: **Może coś mi umknęło, ale właściwie jak chcecie uruchomić testy czegoś, co jeszcze nie istnieje?**

O: Wcale nie chcemy. Nigdy nie pisaliśmy, że należy uruchamiać testy w pierwszej kolejności, pisaliśmy tylko, że tworzenie programu należy zacząć od napisania testów. Zaraz po stworzeniu kodu testującego klasy, które chcemy testować, nie będą jeszcze istnieć; zatem w ogóle nie będziesz w stanie skompilować kodu testującego aż do momentu napisania kodu „pomocniczego”. Dzięki niemu kod będzie można skompilować, jednak sprawdzanie testów nigdy nie zakończy się pomyślnie.

P: **W takim razie wciąż nie widzę sensu takiego postępowania. Dlaczego nie poczekać z tym do momentu zakończenia pisania kodu i stworzyć testy dopiero wtedy, gdy będzie on gotowy?**

O: Proces analizy (i pisania) kodu testowego pomaga w uporządkowaniu wyobrażeń i pomysłów jak powinny wyglądać same metody. Gdy tylko implementacja zostanie zakończona, będziesz mieć kod testujący oczekujący na sprawdzenie, czy jest w porządku. Poza tym doskonale wiesz, że jeśli nie zrobisz tego teraz, to nie zrobisz tego już nigdy. Zawsze znajdziesz się coś bardziej interesującego do zrobienia.

W optymalnym przypadku możesz napisać fragment kodu testującego, a następnie tylko te fragmenty właściwego kodu implementacji, które są konieczne do tego, aby sprawdzanie testów mogło się zakończyć pomyślnie.

Następnie napisz kolejny fragment kodu testującego oraz kolejne fragmenty właściwego kodu implementacji konieczne do pomyślnego przejścia testów. Podczas każdej z takich iteracji należy wykonywać wszystkie wcześniejsze testy, aby mieć pewność, że podczas wprowadzania najnowszych modyfikacji w kodzie nie pojawiły się jakieś błędy.

Kod testowy dla klasy ProstyPortal

```
public class ProstyPortalTester {
    public static void main(String[] args) {
        ProstyPortal wit = new ProstyPortal();
        int[] polozenia = {2,3,4}; ←
        wit.setPolaPolozenia(polozenia);
        String wybranePole = "2"; ←
        String wynik = wit.sprawdz(wybranePole);
        String wynikTestu = "niepowodzenie";
        if (wynik.equals("trafiony")) {
            wynikTestu = "zakończony pomyślnie";
        }
        System.out.println(wynikTestu);
    }
}
```

utworzenie obiektu klasy ProstyPortal

utworzenie tablicy liczb całkowitych określających położenie zatapianego portalu (tablica ta zawiera trzy kolejne liczby całkowite spośród 7 możliwych)

wywołanie metody ustawiającej utworzonego obiektu

symulacja pola wybranego przez gracza

wywołanie metody sprawdz() utworzonego obiektu i przekazanie do niej wybranego pola

jeśli w odpowiedzi na symulowane pole wybrane przez gracza (2) zostanie zwrotny wynik „trafiony”, oznacza to, że metoda działa

zwracamy wynik testu: „niepowodzenie” lub „zakończony pomyślnie”



Zaostrz ołówek

Na kilku kolejnych stronach przedstawiony zostanie proces implementacji klasy ProstyPortal, a dopiero potem ponownie zajmiemy się klasą testującą. Spójrzmy na powyższy kod testujący i zastanówmy się, co jeszcze należałyby do niego dodać. Czego nasza klasa nie testuje, a co powinno zostać sprawdzone? Poniżej napisz swoje pomysły (lub wiersze kodu):

Metoda sprawdz()

Zapewne nie jest to idealny przykład odwzorowania kodu przygotowawczego na normalny kod napisany w Javie, wprowadzimy w nim bowiem jeszcze kilka modyfikacji. Kod przygotowawczy daje nam jednak doskonałe pojęcie odnośnie tego, co powinien robić właściwy kod metody, a naszym zadaniem jest teraz opracowanie takiego kodu napisanego w Javie, który będzie „wiedzieć”, jak te czynności wykonać.

Pisząc poniższy kod, myśl jednocześnie o tych jego fragmentach, które w przyszłości będziesz chciał (lub musiał) poprawić. Fragmenty kodu oznaczone cyframi w kółeczkach, na przykład 1, oznaczają zagadnienia związane zarówno ze składnią Javy, jak i możliwościami tego języka, o których jeszcze nie mówiliśmy. Zostały one omówione na kolejnej stronie.

POBIERZ pole wskazane przez gracza

SKONWERTUJ pole na liczbę typu int

POWTÓRZ dla wszystkich komórek w tablicy typu int

JEŚLI wskazane pole odpowiada jednej z komórek położenia

INKREMENTUJEMY liczbę trafień

// SPRAWDŹ, czy to była ostatnia komórka położenia

JEŚLI ilość trafień wynosi 3

ZWRÓĆ wynik „zatopiony”

W PRZECIWNYM RAZIE Portal nie został zatopiony, więc zwracamy „trafiony”

W PRZECIWNYM RAZIE zwracamy „pułdo”

```
public String sprawdz(String stringPole) {
    1   int strzał = Integer.parseInt(stringPole); ← konwersjałańcucha znaków na liczbę całkowitą
        ↑ utworzenie zmiennej, która będzie przechowywać wynik metody; początkowo jest w niej zapisywanyłańcuch „pułdo” (czyli zakładamy, że gracz spudłował)
    2   String wynik = "pułdo"; ←
        ↑ czynności w pętli będą powtarzane dla każdej komórki tablicy (każdej komórki określającej położenie Portalu)
    for (int pole : polaPołożenia) { ←
        ↑
        if (strzał == pole) { ← porównanie pola podanego przez gracza z tym elementem (komórką) tablicy
            wynik = "trafiony"; ←
            iloscTrafien++; 3 ← gracz trafił!
        4   break; ←
            ↑ przerwanie realizacji pętli, gdyż dalsze sprawdzanie komórek nie jest konieczne
        } // koniec if
    } // koniec for

    if (iloscTrafien == polaPołożenia.length) {
        wynik = "zatopiony"; ←
        ↑ działanie pętli zostało zakończone, ale należy sprawdzić, czy Portal został „zatopiony” (trafiony trzy razy) i w razie czego odpowiednio zmienić wynik
    } // koniec if

    System.out.println(wynik); ←
        ↑ wyświetlenie wyniku („pułdo”, chyba że domyślny wynik został zmieniony na „trafiony” lub „zatopiony”)

    return wynik; ←
        ↑ zwrócenie wyniku do metody wywołującej
} // koniec metody
```

Kilka nowych rzeczy

Na tej stronie zostały przedstawione nowe zagadnienia, z którymi wcześniej jeszcze się nie spotkaliśmy. Ale nie martw się! Pozostałe szczegółowe informacje na temat przedstawionego kodu zostały zamieszczone pod koniec rozdziału. Informacje podane na tej stronie wystarczą Ci do kontynuowania lektury.

1 Konwersja łańcucha znaków na liczbę

klasa dostarczana razem z Java

`Integer.parseInt("3")`

metoda klasy `Integer`, która „wie”, jak przetworzyć łańcuch znaków na liczbę, którą ten łańcuch reprezentuje

pobiera łańcuch znaków

2 Pętla for

Deklarujemy zmienną, która będzie przechowywać jeden element tablicy. Podczas każdej iteracji pętli zmienią ta (która w naszym przypadku nosi nazwę `pole`) będzie zawierać kolejny element tablicy, aż do momentu pobrania każdego z nich punkt 4. poniżej).

`for (int pole : polaPolozenia) {}`

Dwukropki oznacza „w”, a zatem, całe to wyrażenie oznacza: dla każdej wartości całkowitej w tablicy `polaPolozenia`.

Tablica, na której ma operować pętla. Podczas każdej iteracji pętli w zmiennej `pole` będzie umieszczany kolejny element tablicy. (Więcej informacji na ten temat znajdziesz pod koniec tego rozdziału).

3 Operator postinkrementacji

`iloscTrafien++`

`++` oznacza, że do dowolnej bieżącej wartości zmiennej należy dodać 1 (innymi słowy, że wartość zmiennej należy powiększyć o jeden).

Wyrażenie `iloscTrafien++` to prawie to samo (przynajmniej w tym przypadku), co instrukcja `iloscTrafien = iloscTrafien + 1;` (choć skrócony zapis jest nieco bardziej wydajny).

4 Instrukcja break

`break;`

Przerywa działanie pętli. Niezwłocznie. W tym miejscu. Żadnych iteracji, żadnych testów ani warunków, po prostu — koniec!

kod przygotowawczy kod testowy kod właściwy

Nie istnieja
głupie pytania

P: Co się stanie, jeśli do metody Integer.parseInt() zostanie przekazany łańcuch znaków, który nie reprezentuje liczby? Poza tym, czy metoda ta rozpoznaje liczby zapisane słownie, na przykład „trzy”?

O: Metoda Integer.parseInt() operuje wyłącznie na łańcuchach zawierających znaki reprezentujące cyfry (0,1,2,3,4,5,6,7,8,9). Jeśli spróbujesz przekazać do metody łańcuch w stylu „trzy” lub „blup”, to w trakcie działania programu „wyleci w powietrze”. (Przy czym określenie „wyleci w powietrze” oznacza, że zostanie zgłoszony wyjątek. Jednak te zagadnienia będziemy omawiać dopiero w „wyjątkowym” rozdziale. A zatem, jak na razie, określenie, że program „wyleci w powietrze” jest całkiem dobrym przybliżeniem tego co się stanie).

P: Na początku tej książki przedstawiliście przykład pętli for, która była zupełnie inna od użytej w omawianym programie — czy istnieją zatem dwie różne wersje tych pętli?

O: Tak! Od samego początku istnienia języka Java była w nim dostępna jedna wersja pętli for (opisujemy ją w dalszej części tego rozdziału), o następującej postaci:

```
for (int i = 0; i < 10; i++) {  
    // wykonujemy coś 10 razy  
}
```

Można jej używać do definiowania dowolnych, potrzebnych pętli. Jednak... zaczynając od Javy 5.0 (wersji Tiger), kiedy chcemy wykonać jakieś operacje na elementach tablicy (lub dowolnej innej kolekcji, jak przekonasz się *następnym* rozdziałem), to możemy także zastosować rozszerzoną (to oficjalny termin) wersję pętli for. Oczywiście na elementach tablicy można także wykonywać operacje, wykorzystując przy tym zwyczajną, starą pętlę for, jednak wykorzystanie jej rozszerzonej postaci znacznie ułatwia zadanie.

Ostateczny kod klas ProstyPortal oraz ProstyPortalTester

```
public class ProstyPortalTester {  
    public static void main (String[] args) {  
        ProstyPortal wit = new ProstyPortal();  
        int[] polozenia = {2,3,4};  
        wit.setPolaPolozenia(polozenia);  
        String wybranePole = "2";  
        String wynik = wit.sprawdz(wybranePole);  
    }  
  
    class ProstyPortal {  
  
        int [] polaPolozenia;  
        int iloscTrafien;  
  
        public void setPolaPolozenia(int[] ppol) {  
            polaPolozenia = ppol;  
        }  
  
        public String sprawdz(String stringPole) {  
            int strzał = Integer.parseInt(stringPole);  
            String wynik = "pudło";  
            for (int pole : polaPolozenia) {  
                if (strzał == pole) {  
                    wynik = "trafiony";  
                    iloscTrafien++;  
                    break;  
                }  
            } // koniec pętli  
            if (iloscTrafien == polaPolozenia.length) {  
                wynik = "zatopiony";  
            }  
            System.out.println(wynik);  
            return wynik;  
        } // koniec metody  
    } // koniec klasy
```

Co powinniśmy ujrzeć po uruchomieniu powyższego kodu?

Kod testujący tworzy obiekt ProstyPortal i przypisuje mu położenie 2, 3, 4. Następnie przekazuje do wywołania metody sprawdz() łańcuch znaków „2” symulujący współrzędne pola wybranego przez gracza. Zakładając, że kod działa prawidłowo, na ekranie powinien zostać wyświetlony komunikat:

```
java ProstyPortalTester  
trafiony
```

Gdzieś w powyższym kodzie przyplatał się mały błąd. Program się kompliuje i działa, jednak czasami... Póki co nie przejmuj się tym, jednak kiedyś będziemy musieli się tym zająć.

kod przygotowawczy

kod testowy

kod właściwy



Zaostrz ołówek

Napisaliśmy już klasę testującą oraz klasę ProstyPortal, jednak wciąż nie napisaliśmy samej gry. Dysponując kodami przedstawionymi na poprzedniej stronie oraz specyfikacją samej gry, masz za zadanie napisać, jak według Ciebie powinien wyglądać kod przygotowawczy klasy gry. Podaliśmy poniżej kilka wierszy, aby ułatwić Ci pracę. Kod klasy gry został przedstawiony na następnej stronie, zatem **nie odwracaj kartki, zanim nie skończysz ćwiczenia!**

Powinieneś uzyskać nie mniej niż 12 i nie więcej niż 18 wierszy kodu (włącznie z tymi przedstawionymi przez nas, jednak *nie uwzględniając* wierszy, w których znajdują się wyłącznie nawiasy klamrowe).

METODA public static void main(String[] args)

ZADEKLARUJ zmienną typu int przechowującą liczbę ruchów gracza, nadaj jej nazwę iloscRuchow

Oto czynności, jakie powinna wykonywać klasa ProstyPortalGra

1. Tworzyć jeden obiekt klasy ProstyPortal.
2. Określić jego położenie (spośród siedmiu wirtualnych komórek wybrać trzy sąsiadujące ze sobą).
3. Prosić gracza o podanie pola.
4. Sprawdzić pole wskazane przez gracza.
5. Powtarzać czynności aż do zatopienia Portalu.
6. Wyświetlić graczowi informacje o liczbie wykonanych ruchów.

WYZNACZ liczbę losową z zakresu od 0 do 4, która określi początek miejsca, w którym zostanie umieszczony Portal

DOPÓKI Portal nie został zatopiony

POBIERZ dane wejściowe wpisane przez gracza w wierszu poleceń

Przykład pełnej rozgrywki

```
ca Wiersz poleceń
T:\>java ProstyPortalGra
Podaj liczbę 3
pułko
Podaj liczbę 4
trafiony
Podaj liczbę 5
trafiony
Podaj liczbę 6
zatopiony
4 ruchów
```

kod przygotowawczy

kod testowy

kod właściwy

Kod przygotowawczy klasy ProstyPortalGra

Wszystko dzieje się w metodzie main()

W niektóre rzeczy po prostu musisz uwierzyć. Na przykład w poniższym kodzie przygotowawczym jest wiersz o treści: „POBIERZ dane wprowadzane przez gracza w wierszu poleceń”. Trzeba zaznaczyć, że ta operacja to nieco zbyt dużo niż chcielibyśmy w całości sami implementować. Lecz na szczęście używamy obiektowego języka programowania. A to oznacza, że możesz poprosić jakąś *inną* klasę lub obiekt o wykonanie pewnych operacji bez konieczności przejmowania się tym, *jak* je należy zrealizować. Pisząc kod przygotowawczy, powinieneś założyć, że w *jakiś* sposób będziesz w stanie wykonać wszystkie niezbędne operacje, dzięki temu wszystkie siły możesz włożyć w opracowanie logiki działania kodu.

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

ZADEKLARUJ zmienną typu int przechowującą ilość ruchów gracza, nadaj jej nazwę `iloscRuchow` i przypisz wartość 0.

UTWÓRZ nowy obiekt klasy `ProstyPortal`.

WYZNACZ liczbę losową z zakresu od 0 do 4, która określi początek miejsca, w którym zostanie umieszczony Portal.

UTWÓRZ tablicę trzech liczb całkowitych, z których pierwsza będzie wartością wygenerowaną losowo, druga wartością o jeden większą, a trzecia — wartością większą o 2 (na przykład: 2, 3 i 4).

WYWOŁAJ metodę `setPolaPolozienia()` obiektu `ProstyPortal`.

ZADEKLARUJ zmienną logiczną reprezentującą stan gry i nadaj jej nazwę `czyIstnieje`. **PRZYPISZ** tej zmiennej wartość początkową `true`.

DOPÓKI Portal istnieje (czyli `czyIstnieje == true`):

POBIERZ dane wejściowe wpisane przez gracza w wierszu poleceń

`// SPRAWDŹ` pole wskazane przez gracza

WYWOŁAJ metodę `sprawdz()` obiektu klasy `ProstyPortal`

INKREMENTUJ zmienną `iloscRuchow`

`// SPRAWDŹ`, czy Portal został zatopiony

JEŚLI zwrócony został wynik „zatopiony”:

PRZYPISZ zmiennej `czyIstnieje` wartość `false` (co oznacza, że pętla nie zostanie już więcej wykonana)

WYŚWIETL ilość ruchów gracza

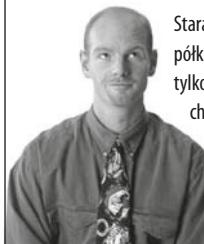
KONIEC JEŚLI

KONIEC DOPÓKI

KONIEC METODY

Metapoznaniowa odpowiedź

Staraj się nie pracować przez zbyt długi okres czasu tylko jedną półkulą mózgu. Praca lewą półkulą mózgu przez 30 minut to tak jakby przez 30 minut wykonywać pracę fizyczną tylko lewą ręką. Regularnie zmieniaj używanie półkul, zapewniając im w ten sposób chwilę odpoczynku. Gdy zmieniasz używaną półkulę, ta druga odpoczywa i może się zregenerować. Lewa półkula mózgu odpowiada za takie operacje jak sekwencje czynności czy rozwiązywanie i analizę problemów logicznych. Z kolei do zadań prawej półkuli mózgu należą: pojmowanie metafor, kreatywne rozwiązywanie problemów, dopasowywanie wzorców oraz wizualizacja.





CELNE SPOSTRZEŻENIA

- Program pisany w Javie powinien zaczynać się od projektu sporządzanego na wysokim poziomie abstrakcji.
- Zazwyczaj tworząc nową klasę, zaczniesz od napisania trzech wersji kodu:
 - kodu przygotowawczego**
 - kodu testowego**
 - właściwego kodu (w Javie)**
- Kod przygotowawczy powinien opisywać *co*, a nie *jak*, należy wykonać. Implementacją będziesz się zajmował później.
- Wykorzystaj kod przygotowawczy jako pomoc przy tworzeniu kodu testującego.
- Napisz kod testujący, *zanim* zaczniesz implementować metody.

- Jeśli wiesz, ile razy należy powtórzyć wykonanie kodu umieszczonego wewnątrz pętli, to staraj się korzystać z pętli **for**, a nie z pętli **while**.
- W celu dodania wartości 1 do zmiennej używaj operatora *preinkrementacji* lub *postinkrementacji* (`x++`).
- W celu odjęcia wartości 1 od zmiennej używaj operatora *predekrementacji* lub *postdekrementacji* (`x--`).
- Używaj metody `Integer.parseInt()` w celu przetworzenia łańcucha znaków na liczbę całkowitą.
- Metoda `Integer.parseInt()` działa wyłącznie w przypadku, gdy przekazany do niej łańcuch znaków zawiera cyfry (na przykład: „1”, „2” i tak dalej).
- Posłuż się instrukcją *break*, aby wcześniej zakończyć wykonywanie pętli (na przykład nawet w przypadku, gdy test logiczny wciąż jest spełniony).



Metoda main() gry

Podobnie jak podczas tworzenia klasy ProstyPortal także i teraz myśl o tych fragmentach kodu klasy, które w przyszłości będziesz chciał (lub musiały) poprawić. Fragmenty kodu oznaczone cyframi w kóleczkach, na przykład 1, zwierają coś, co chcemy wyróżnić. Zagadnienia te zostały dokładniej opisane na kolejnej stronie. Jeśli zastanawiasz się, dlaczego pominęliśmy etap tworzenia kodu testującego dla tej klasy, to spieszmy wyjaśnić, że w przypadku samej gry klasa testująca nie będzie potrzebna. Tworzona klasa ma tylko jedną metodę, cóż zatem mielibyśmy umieścić w klasie testującej? Tworzyć osobną klasę tylko po to, aby wywołać metodę main() tej klasy? To chyba przesada.

ZADEKLARUJ zmienną typu int przechowującą liczbę ruchów gracza i przypisz wartość 0

UTWÓRZ nowy obiekt klasy ProstyPortal

WYZNACZ liczbę losową z zakresu od 0 do 4

UTWÓRZ tablicę trzech liczb całkowitych i

WYWOŁAJ metodę setPolaPolozenia() obiektu ProstyPortal

ZADEKLARUJ zmienną logiczną czyIstnieje

DOPÓKI Portal istnieje

POBIERZ dane wejściowe wpisane przez gracza

//SPRAWDŹ je

WYWOŁAJ metodę sprawdz() obiektu ProstyPortal

INKREMENTUJ zmienną iloscRuchow

JEŚLI zwrócony został wynik „zatopiony”

PRZYPISZ zmiennej czyIstnieje wartość false

WYSWIETL liczbę ruchów gracza

```
public static void main(String[] args) {
    int iloscRuchow = 0; 1 tworzymy zmienną, która będzie określić, ile ruchów wykonat gracz
    PomocnikGry pomocnik = new PomocnikGry(); to specjalna napisana przez nas klasa, która posiada metodę pobierającą dane wejściowe podawane przez gracza i udaje, że jest częścią języka Java
    ProstyPortal portal = new ProstyPortal(); 2 tworzymy obiekt ProstyPortal
    int liczbaLosowa = (int) (Math.random() * 5); generujemy liczbę losową, która będzie zapisana w pierwszej komórce tablicy i na jej podstawie określamy zawartość tablicy polożenia
    int[] polozenie = {liczbaLosowa, liczbaLosowa+1, liczbaLosowa+2};
    portal.setPolaPolozenia(polozenie); przekazujemy do obiektu ProstyPortal jego położenie (w formie tablicy)
    boolean czyIstnieje = true; tworzymy zmienną logiczną określającą, czy gra wciąż powinna się toczyć; jest używana w warunku pętli while
    while (czyIstnieje == true) { 2
        String pole = pomocnik.pobierzDaneWejsciowe("Podaj liczbę"); pobieramy ciąg znaków podany przez gracza
        String wynik = portal.sprawdz(pole); prosimy obiekt ProstyPortal, aby sprawdzić pole podane przez gracza, zwrócony wynik jest zapisywany jako ciąg znaków
        iloscRuchow++; inkrementujemy wartość zmiennej określającej ilość ruchów
        if (wynik.equals("zatopiony")) { A co, jeśli nasz Portal zostanie „zatopiony”? W takim przypadku przypisujemy false (dzięki czemu zawartość pętli nie zostanie już więcej wykonana) i wyświetlamy liczbę ruchów.
            czyIstnieje = false; zatrzymujemy pętlę while
            System.out.println(iloscRuchow + " ruchów");
        } // koniec if
    } // koniec while
} // koniec main
```

random() i pobierzDaneWejsciowe()

Na tej stronie przedstawione zostały dwa zagadnienia wymagające dodatkowych wyjaśnień. Wyjaśnienia te będą krótkie — zapewniają Ci jedynie możliwość dalszej lektury; bardziej pełne informacje o klasie PomocnikGry zostały zamieszczone pod koniec rozdziału.

① Generacja liczby losowej

```
int liczbaLosowa = (int) (Math.random() * 5)
```

Deklarujemy zmienną typu int, która będzie przechowywać wygenerowaną liczbę losową.

To jest „rzutowanie” — wymusza ono, aby coś, co zostało podane bezpośrednio za operatorem rzutowania, przyjęło wartość typu określonego w rzutowaniu (czyli w nawiasach). Metoda Math.random() zwraca liczbę typu double, zatem musimy rzutować ją na typ int (gdys potrzebna nam jest liczba całkowita z zakresu od 0 do 4). W tym przypadku w wyniku rzutowania zostanie usunięta część utamkowa liczby zmiennoprzecinkowej.

Metoda Math.random() zwraca liczbę większą lub równą 0 i mniejszą od 1. A zatem cały wzór (razem z rzutowaniem) zwraca liczbę z zakresu od 0 do 4 (konkretnie rzecz biorąc, zwracana jest wartość z zakresu 0 - 4.9999..., przy czym część utamkowa zostaje odrzucona).

Klasa dostarczana wraz z Javą.
Metoda klasy Math.

② Pobieranie danych wejściowych wprowadzanych przez gracza, za pomocą użycia klasy PomocnikGry

```
String pole = pomocnik.pobierzDaneWejsciowe("Podaj liczbę");
```

Deklarujemy zmienną typu String, która będzie służyć do przechowywania pobranego ciągucha znaków wpisanego przez gracza (na przykład: „2”, „5” itp.).

Utworzona wcześniej kopia klasy, której zadaniem jest pomaganie w obsłudze gry. Klasa ta nosi nazwę PomocnikGry i jeszcze nie została przedstawiona (choć niedługo już ją poznasz).

Metoda wymaga podania ciągucha znaków, który stanowi treść komunikatu wyświetlanego w wierszu poleceń. Dowolny ciąguch, jaki przekażesz w wywołaniu tej metody, zostanie wyświetlony w wierszu poleceń, zanim metoda zacznie oczekiwania informacji wprowadzane przez gracza.

Metoda klasy PomocnikGry, która prosi użytkownika o podanie danych w wierszu poleceń, odczytuje je, kiedy użytkownik naciśnie klawisz Enter i zwraca w formie ciągucha znaków (obiektu typu String).

Ostatnia klasa — PomocnikGry

Napisaliśmy już klasę *portalu*.

Napisaliśmy także klasę *gry*.

Pozostaje nam zatem do napisania jedynie klasa pomocnicza

— ta, która dysponuje metodą `pobierzDaneWejsciowe()`.

Kod obsługujący pobieranie danych z wiersza poleceń jest zbyt skomplikowany, abyśmy chcieli go teraz wyjaśniać. Musielibyśmy poruszać zbyt wiele zagadnień, które lepiej zostawić na później. (Na później, czyli do rozdziału 14.).



Po prostu skopiuj* poniższy kod i skompiluj go.

Umieść wszystkie trzy klasy (`ProstyPortal`, `ProstyPortalGra` oraz `PomocnikGry`) w tym samym folderze i przejdź do niego (aby stał się bieżącym folderem roboczym).



Za każdym razem, gdy zobaczysz ikonę , wiedz, że przedstawiony poniżej kod należy przepisać bez żadnych zmian i wierzyć, że robi dokładnie to, co trzeba. Zaufaj mu. *Później* dowiesz się, jak on działa.



Kod gotowy do użycia

```
import java.io.*;
public class PomocnikGry {
    public String pobierzDaneWejsciowe(String komunikat) {
        String wierszWej = null;
        System.out.print(komunikat + " ");
        try {
            BufferedReader sw = new BufferedReader(
                new InputStreamReader(System.in));
            wierszWej = sw.readLine();
            if (wierszWej.length() == 0) return null;
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
        return wierszWej;
    }
}
```

* Doskonale wiemy, jak bardzo lubisz stukać na klawiaturze, jednak na wypadek tych sporadycznych sytuacji, kiedy będziesz mieć coś ciekawszego do roboty, przygotowaliśmy „Gotowy kod”, który można pobrać z lokalizacji <ftp://ftp.helion.pl/przyklady/javrg2.zip>

Zagrajmy

Oto co się stanie, kiedy wpiszemy liczby: 1, 2, 3, 4, 5 oraz 6. Wygląda na to, że wszystko jest w porządku.

Pełny przebieg gry

(Twoje wyniki mogą być inne)

```
c:\ Wiersz polecenia
T:\>java ProstyPortalGra
Podaj liczbę 1
pudło
Podaj liczbę 2
pudło
Podaj liczbę 3
pudło
Podaj liczbę 4
trafiony
Podaj liczbę 5
trafiony
Podaj liczbę 6
zatopiony
6 ruchów
```

Ale co to? Błąd?

O rany!

Oto co się stanie, kiedy wpiszemy liczby: 2, 2 i 2.

Inny przykład przebiegu gry

(w którym można zauważać błąd)

```
c:\ Wiersz polecenia
T:\>java ProstyPortalGra
Podaj liczbę 2
trafiony
Podaj liczbę 2
trafiony
Podaj liczbę 2
zatopiony
3 ruchów
```



Zaostrz ołówek

Cóż za napięcie!

Czy **znajdziemy** błąd?

Czy go **poprawimy**?

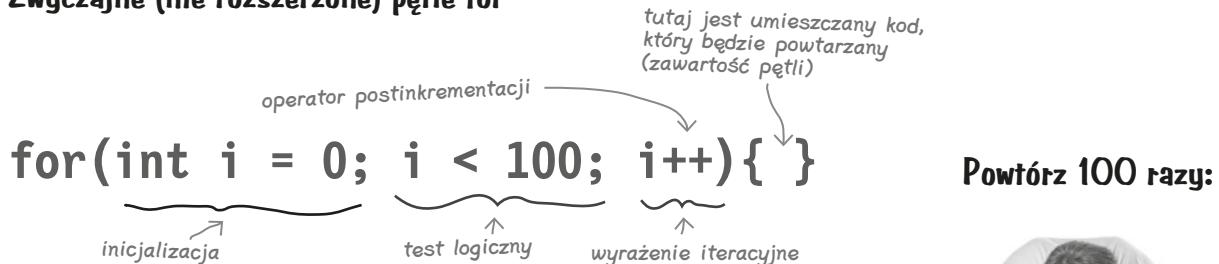
Nie rozpraszać się i przeczytaj kolejny rozdział, w którym znajdziesz, między innymi, odpowiedzi na powyższe pytania.

A na razie sprawdź, czy będziesz w stanie wpaść na pomysł, w czym tkwi problem i jak go poprawić.

Trochę więcej o pętlach for

Przedstawiliśmy już cały kod, który miał się znaleźć w tym rozdziale (jednak wróćmy do niego w kolejnym — szóstym — rozdziale, aby stworzyć pełną wersję naszej gry). Nie chcieliśmy jednak przerывать Ci pracy nad kodem gry, aby podać poniższe dodatkowe informacje. Zaczniemy od przedstawienia dodatkowych informacji dotyczących pętli for; jeśli jesteś programistą używającym języka C++, to kilka ostatnich stron tego rozdziału możesz pominąć.

Zwyczajne (nie rozszerzone) pętle for



Co to oznacza „po polsku”: „Powtórz sto razy”?

Kompilator rozumie to tak:

- * utwórz zmienną `i`, po czym przypisz jej wartość 0;
- * powtarzaj pętlę dopóki wartość zmiennej `i` jest mniejsza od 100;
- * pod koniec każdej iteracji pętli powiększ wartość zmiennej `i` o 1.

Część pierwsza: Inicjalizacja

Tej części pętli możesz używać do zadeklarowania i inicjalizacji zmiennej, której następnie będziesz mógł używać w zawartości pętli. W przeważającej większości przypadków zmienią ta będzie pełnić funkcję licznika. W rzeczywistości w tej części pętli można zainicjalizować więcej niż jedną zmienną; zagadnienie to przedstawimy jednak w dalszej części książki.

Część druga: test logiczny

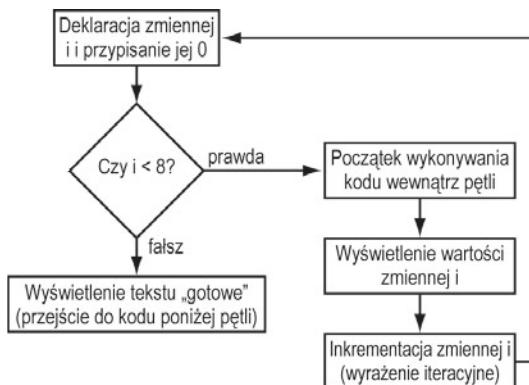
W tym miejscu jest wykonywane sprawdzenie warunku logicznego. Jakikolwiek kod zostanie tu umieszczony, musi on w wyniku zwrócić wartość logiczną (wiesz, o co chodzi — **true** lub **false**). Można wykorzystać zwyczajny warunek logiczny, taki jak `(x >= 4)`, a nawet wywołać metodę, która zwraca wartość logiczną.

Część trzecia: wyrażenie iteracyjne

W tym miejscu można umieścić jedną lub kilka operacji, które mają być wykonywane podczas każdej iteracji pętli. Pamiętaj jednak, że wykonanie tych operacji następuje *pod koniec* każdej iteracji.

Wycieczki po pętli

```
for (int i = 0; i < 8; i++) {
    System.out.println(i);
}
System.out.println("gotowe");
```



Różnica pomiędzy pętlami for i while

Pętla while posiada jedynie test logiczny, nie daje natomiast możliwości określenia inicjalizacji ani wyrażenia iteracyjnego. Pętle tego typu są najwygodniejsze w sytuacjach, kiedy nie wiadomo, ile razy pętla ma zostać wykonana, a umieszczone w niej operacje mają być realizowane tak długo, jak długo podany warunek jest spełniony. Jednak jeśli *wiemy*, ile razy należy wykonać pętlę (na przykład 7 razy lub tyle razy, ile jest komórek w tablicy), to znaczenie bardziej przejrzysta jest pętla for. Oto przedstawiona powyżej pętla zapisana przy wykorzystaniu instrukcji while:

```
int i = 0;           ← musimy zadeklarować i określić
                    ← wartość początkową licznika
while (i < 8) {
    System.out.println(i);
    i++;             ← musimy inkrementować
                    ← wartość licznika
}
System.out.println("gotowe");
```

Wyniki

A screenshot of a terminal window titled "Wiersz polecenia". The command "java Test" is entered, followed by several numbers from 0 to 7, and finally the text "gotowe".

++ --

Operatory pre i post inkrementacji oraz dekrementacji

Skrócony sposób dodawania lub odejmowania wartości 1 od bieżącej wartości zmiennej.

x++;

Odpowiada:

x = x + 1;

W powyższym kontekście oba przedstawione przykłady mają to samo znaczenie:

„dodaj 1 do bieżącej wartości zmiennej *x*” lub „**inkrementuj** wartość *x* o 1”.

Z kolei:

x--;

odpowiada:

x = x - 1;

Oczywiście byłoby nieprawdopodobne, gdyby to było wszystko. Otóż miejsce zapisu operatora (przed lub za zmienną) ma wpływ na wynik. Umieszczenie go przed zmienną (na przykład **++x**) oznacza: „*najpierw inkrementuj x o 1, a dopiero potem użij wartości zmiennej*”. Sposób zapisu operatora ma znaczenie wyłącznie w przypadku, gdy **++x** jest częścią jakiegoś większego wyrażenia, a nie stanowi pojedynczej instrukcji.

int x = 0;

int z = ++x;

W efekcie obie zmienne mają wartość 1.

Jednak umieszczenie operatora **++** za zmienną *x* daje inne wyniki:

int x = 0;

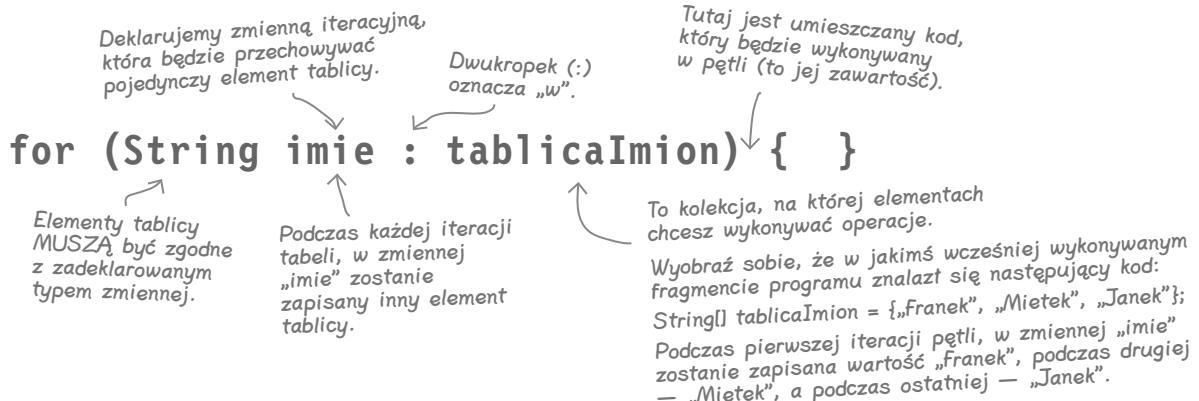
int z = x++;

W tym przypadku *x* ma wartość 1, lecz **z ma wartość 0!** Dzieje się tak, gdyż zmiennej *z* najpierw jest przypisywana wartość zmiennej *x*, a dopiero potem wartość *x* zostaje powiększona o 1.

Rozszerzone pętle

Zaczynając od wersji 5.0 (Tiger) język Java udostępnia drugi rodzaj pętli for, określany jako pętla *rozszerzona*. Te rozszerzone pętle for ułatwiają wykonywanie operacji na elementach tablic oraz wszelkich rodzajach kolekcji (o *innym* rodzaju kolekcji dowiesz się w następnym rozdziale).

Tak naprawdę, ten nowy rodzaj pętli nie oferuje niczego więcej; niemniej jednak, zważywszy, że jest to najczęstsze zastosowanie pętli for, to i tak warto go było dodać do języka. Przyjrzymy mu się dodatkowo także w następnym rozdziale, przy okazji prezentacji kolekcji, które *nie są* tablicami.



Co to oznacza „po polsku”: „Dla każdego elementu w tablicy tablicaImion, przypisz wartość elementu zmiennej „imie” i wykonaj zawartość pętli.”

Jak to widzi kompilator:

- * Utwórz zmienną typu String o nazwie imie i przypisz jej początkowo wartość null.
- * Zapisz w zmiennej imie pierwszą wartość z tablicy tablicaImion.
- * Wykonaj zawartość pętli (kod umieszczony pomiędzy jej nawiasami klamrowymi).
- * Pobierz kolejną wartość tablicy tablicaImion i zapisz ją w zmiennej imie.
- * Powtarzaj dwie ostatnie czynności *tak długo jak w tablicy są jakieś elementy do odczytania*.

Uwaga: programiści, którzy wcześniej używali innych języków programowania, mogą nazywać tą rozszerzoną pętlę for, inaczej: „for each” lub „for in”. Odpowiada to rzeczywistości, gdyż należy ją rozumieć jako: „dla KAZDEGO elementu W kolekcji...”.

Część pierwsza: deklaracja zmiennej iteracyjnej

Używaj tego fragmentu instrukcji by zadeklarować i zainicjalizować zmienną, której następnie będziesz używał wewnętrz pętli. Podczas każdej iteracji pętli, zmienna ta będzie zawierać kolejny element pobrany z kolekcji. Typ tej zmiennej musi być zgodny z typem elementów tablicy! Na przykład, nie możesz zadeklarować zmiennej iteracyjnej typu int, jeśli tablica zawiera łańcuchy znaków (String[]).

Część druga: kolekcja

Ta część pętli musi być odwołaniem do tablicy lub innej kolekcji. Powtarzam — póki co nie przejmuj się tymi „innymi typami” kolekcji — poznasz je w następnym rozdziale.

Konwersja łańcucha znaków na liczbę całkowitą typu int

```
int pole = Integer.parseInt(stringPole);
```

Gdy gracz zostanie o to poproszony, wpisuje numer pola w wierszu polecień. Numer ten zostaje odczytany przez program jako łańcuch znaków („2”, „0” i tak dalej), i jest następnie przekazywany do metody sprawdz().

Jednak z punktu widzenia programu numery pól to liczby całkowite będące indeksami komórek tablicy, a zatem nie można ich bezpośrednio porównywać z łańcuchami znaków.

Na przykład, **poniższa instrukcja jest nieprawidłowa:**

```
String liczba = "2";
int x = 2;
if (x == liczba) // straszliwy wybuch!
Próba skompilowania takiego fragmentu kodu sprawi, że kompilator zacznie się śmiać i kpić z Ciebie:
test.java:5: operator == cannot be
applied to int,java.lang.String
    if (x == liczba) {}
        ^
```

A zatem, aby rozwiązać ten problem, musimy zamienić łańcuch znaków „2” na wartość 2 typu int. Biblioteka klas Java zawiera klasę o nazwie Integer (przy czym jest to *klasa*, a nie jeden z *podstawowych typów danych*), której jednym z zadań jest zamienianie łańcuchów znaków *reprezentujących* liczby na *faktyczne* liczby.

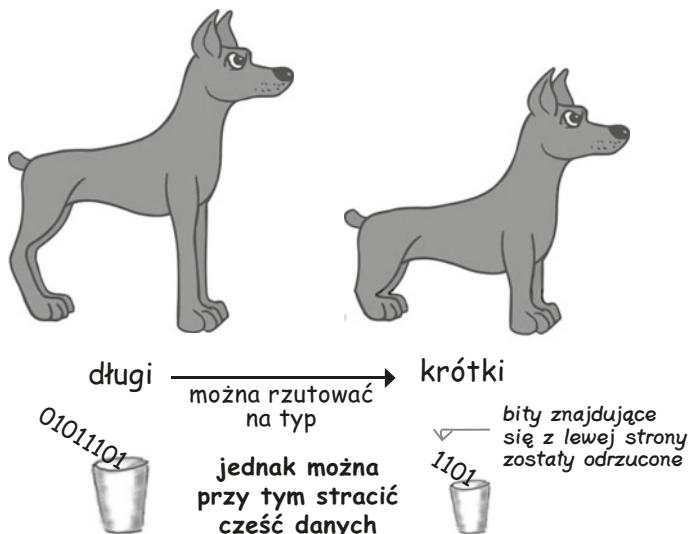
klasa dostarczana
wraz z Java

metoda wymaga
przekazania
łańcucha znaków

Integer.parseInt("3");

metoda klasy Integer, która „wie”,
w jaki sposób zamieniać łańcuchy
znaków reprezentujące liczby na
faktyczne liczby

Rzutowanie wartości typów podstawowych



W trzecim rozdziale opisane zostały zagadnienia związane z wielkościami różnych typów podstawowych; wspomniliśmy także o tym, że nie można umieścić danej „większego” typu w zmiennej „mniejszego” typu:

```
long y = 42;
int x = y; // błąd kompilacji
```

Typ long jest większy od typu int, a kompilator nie wie, „gdzie” ta konkretna zmienna „była wcześniej”. Być może bawiła się gdzieś z innymi zmiennymi typu long i zawiera naprawdę dużą wartość. Aby zmusić kompilator do zapisania wartości większego typu podstawowego w zmiennej mniejszego typu, należy użyć operatora **rzutowania**. Operator ten przedstawiony został na poniższym przykładzie:

```
long y = 42; // jak na razie wszystko jest w porządku
int x = (int) y; // x = 42 - super!
```

Użycie operatora rzutowania informuje kompilator, że powinien pobrać wartość zmiennej y, skrócić ją do wielkości odpowiadającej typowi int i to, co z niej zostanie, zapisać w zmiennej x. Jeśli wartość zmiennej y była większa od maksymalnej możliwej wartości zmiennej x, to rezultat rzutowania może być dziwny (choć możliwy od obliczenia*):

```
long y = 40002;           // liczba 40002 przekracza zakres
                           // 16-bitowych wartości typu short
short x = (short) y;     // x ma wartość -25534!
```

Niemniej jednak najważniejsze jest to, że kompilator pozwolił nam na wykonanie powyższych operacji. A teraz założmy, że dysponujesz wartością zmiennoprzecinkową i chcesz pobrać jedynie jej część całkowitą (jako wartość typu int):

```
float f = 3.14f;
int x = (int) f; // x będzie mieć wartość 3
```

Nawet nie myśl o rzutowaniu czegokolwiek na wartość logiczną (typu boolean) lub wartości logicznej na liczbę — po prostu o tym zapomnij.

* Wyznaczenie tej wartości wymaga uwzględnienia bitów znaku, „dopełnienia dwójkowego” i innych dziwnych operacji, które zostały bardziej szczegółowo opisane na początku dodatku B.



BĄDŹ JVM



Plik przedstawiony na tej stronie reprezentuje kompletny kod źródłowy programu napisanego w Javie. Twoim zadaniem jest stać się wirtualną maszyną Java i określić, jakie będą wyniki jego wykonania.

```
class Wyniki {  
    public static void main(String[] args) {  
        Wyniki w = new Wyniki();  
        w.doDziela();  
    }  
  
    void doDziela() {  
        int y = 7;  
        for (int x = 1; x < 8; x++) {  
            y++;  
            if (x > 4) {  
                System.out.print(++y + " ");  
            }  
            if (y > 14) {  
                System.out.println(" x = " + x);  
                break;  
            }  
        }  
    }  
}
```

```
T:\>java Wyniki  
12 14
```

-lub-

```
T:\>java Wyniki  
12 14 x = 6
```

-lub-

```
T:\>java Wyniki  
13 15 x = 6
```



Ćwiczenie



Magnesiki z kodem

Działający program Java został podzielony na fragmenty, zapisany na małych magnesach, które przyczepiono do lodówki. Czy jesteś w stanie złożyć go z powrotem w jedną całość, tak aby wygenerował przedstawione poniżej wyniki? Niektóre nawiasy klamrowe spadły na podłogę i były zbyt małe, aby można je było podnieść, dlatego w razie potrzeby możesz je dodawać!

`x++;`

`if (x == 1) {`

`System.out.println(x + " " + y);`

`class PetleFor {`

`for (int y = 4; y > 2; y--) {`

`for (int x = 0; x < 4; x++) {`

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

```
Wiersz polecenia
I:\>java PetleFor
0 4
0 3
1 4
1 3
3 4
3 3
```

Pomieszane komunikaty



Pomieszane komunikaty

Poniżej zamieszczono prosty program w Javie. Brakuje w nim jednego fragmentu. Twoim zadaniem jest **dopasowanie proponowanego bloku kodu** (przedstawionego poniżej) z **wynikami**, które program wygeneruje po wstawieniu wybranego bloku. Nie wszystkie wiersze wyników zostaną wykorzystane, a niektóre z nich mogą być wykorzystane więcej niż jeden raz. Narysuj linie łączące bloki kodu z odpowiadającymi im wynikami. (Wszystkie odpowiedzi można znaleźć na końcu rozdziału).

```
class PomieszanePetleFor {  
    public static void main(String[] args) {  
        int x = 0;  
        int y = 30;  
        for (int zewn = 0; zewn < 3; zewn++) {  
            for (int wewn = 4; wewn > 1; wewn--) {  
  
                } ← tutaj umieść proponowany  
                fragment kodu  
                y = y - 2;  
                if (x == 6) {  
                    break;  
                }  
                x = x + 3;  
            }  
            y = y - 2;  
        }  
        System.out.println(x + " " + y);  
    }  
}
```

Proponowane fragmenty kodu:

x = x + 3;

x = x + 6;

x = x + 2;

x++;

x--;

x = x + 0;

Możliwe dane wynikowe:

45 6

36 6

54 6

60 10

18 6

6 14

12 14

dopasuj każdy z proponowanych fragmentów kodu z jednym z możliwych wyników



Ćwiczenie rozwiążanie

Bądź JVM

```
class Wyniki {
    public static void main(String[] args) {
        Wyniki w = new Wyniki();
        w.doDziela();
    }

    void doDziela() {
        int y = 7;
        for (int x = 1; x < 8; x++) {
            y++;
            if (x > 4) {
                System.out.print(++y + " ");
            }
            if (y > 14) {
                System.out.println(" x = " + x);
                break;
            }
        }
    }
}
```

Czy pamiętasz, aby uwzględnić instrukcję break? Jaki ma ona wpływ na wyniki działania programu?

```
T:\>java Wyniki
13 15  x = 6
```

Magnesiki z kodem

```
class PetleFor {
    public static void main(String[] args) {
        for (int x = 0; x < 4; x++) {
            for (int y = 4; y > 2; y--) {
                System.out.println(x + " " + y);
            }
            if (x == 1) {
                x++;
            }
        }
    }
}
```

Co by się stało, gdyby ten fragment kodu został umieszczony przed pętlą for sterowaną przy użyciu zmiennej y?

Wiersz polecenia

```
T:\>java PetleFor
0 4
0 3
1 4
1 3
3 4
3 3
```

Ćwiczenie – rozwiążanie



Rozwiążanie zagadki

Proponowane fragmenty kodu:

`x = x + 3;`

`x = x + 6;`

`x = x + 2;`

`x++;`

`x--;`

`x = x + 0;`

Możliwe dane wynikowe:

`45 6`

`36 6`

`54 6`

`60 10`

`18 6`

`6 14`

`12 14`

6. Poznaj Java API

Korzystanie z biblioteki Javy



Java jest wyposażona w setki gotowych do użycia klas. Jeśli tylko potrafisz znaleźć w bibliotece Javy, nazywanej **Java API**, to, czego Ci potrzeba, to nie będziesz musiał ponownie wymyślać koła. *W końcu masz ciekawsze rzeczy do roboty.* Jeśli musisz napisać kod, to również dobrze możesz napisać *wyłącznie* te jego fragmenty, które są unikalne dla tworzonej aplikacji. Czy znasz tych programistów, którzy zawsze wychodzą z pracy punktualnie o godzinie 17:00? I tych, którzy nigdy nie pojawiają się przed 10 rano? **Oni używają Java API.** I za jakieś osiem stron także Ty będziesz korzystać z tej biblioteki. A zatem — podstawowa biblioteka Javy to ogromny zbiór klas, które tylko czekają, abyś zastosował je jako elementy budulcowe i abyś zaczął tworzyć swoje programy, posługując się gotowymi fragmentami kodu. Zamieszczone w tym rozdziale przykłady oznaczone jako „Kod gotowy do użycia” to fragmenty kodu, których nie musisz tworzyć sam od podstaw, choć musisz je samodzielnie umieścić w kodzie programu. Java API składa się z fragmentów kodu, których nawet nie musisz wpisywać. Jedyne, co musisz zrobić, to nauczyć się z nich korzystać.

W naszym programie wciąż jest błąd

**Ostatni rozdział zakończył się
w dramatycznych okolicznościach
— w programie znaleźliśmy błąd**

Jak powinien wyglądać przebieg gry?

Oto co się stanie, jeśli po uruchomieniu naszej gry wpiszemy cyfry 1, 2, 3, 4, 5 i 6. Wygląda na to, że wszystko jest w porządku.

Pełny przebieg gry (Twoje wyniki mogą być inne)

```
C:\ Wiersz polecenia

T:\>java ProstyPortalGra
Podaj liczbę 1
pudo
Podaj liczbę 2
pudo
Podaj liczbę 3
pudo
Podaj liczbę 4
trafiony
Podaj liczbę 5
trafiony
Podaj liczbę 6
zatopiony
6 ruchów
```

Jak się objawia błąd?

Oto co się stanie, gdy wpiszemy cyfry 2, 2 i 2.

Przykład innego przebiegu gry (uwidaczniający błąd)

```
C:\ Wiersz polecenia

T:\>java ProstyPortalGra
Podaj liczbę 2
trafiony
Podaj liczbę 2
trafiony
Podaj liczbę 2
zatopiony
3 ruchów
```

**W aktualnej wersji gry, jeśli trafisz, to,
aby założyć portal, wystarczy powtórzyć
ten sam ruch jeszcze dwa razy!**

Co się zatem stało?

Oto co jest źle. Za każdym razem, gdy użytkownik wskaże komórkę, w jakiej znajduje się portal, doliczamy nowe trafienie, a robimy tak nawet w przypadku, gdy wskazana komórka została już „trafiona” wcześniej!

Musimy znaleźć sposób, aby wiedzieć, czy komórka wskazana przez użytkownika nie została już wcześniej „trafiona”. Jeśli została, nie powinniśmy jej liczyć jako nowego trafienia.

```
public String sprawdz(String stringPole) {
    int strzał = Integer.parseInt(stringPole); ← Konwersjałańcucha znaków na liczbę całkowitą.

    String wynik = "pułdo"; ← Utworzenie zmiennej, która będzie przechowywać wynik metody; początkowo jest w niej zapisywanyłańcuch „pułdo” (czyli zakładamy, że gracz spudłował).

    for (int pole : polaPolozenia) { ← Czynności w pętli będą powtarzane dla każdej komórki tablicy.

        if (strzał == pole) {
            wynik = "trafiony"; ← Porównanie pola podanego przez gracza z tym elementem (komórką) tablicy.

            iloscTrafien++; ← Gracz trafiał

            break; ← Przerwanie realizacji pętli, gdy dalsze sprawdzanie komórek nie jest konieczne.

        } // koniec if
    } // koniec for

    if (iloscTrafien == polaPolozenia.length) {
        wynik = "zatopiony"; ← Działanie pętli zostało zakończone, ale należy sprawdzić, czy Portal został „zatopiony” (trafiony trzy razy), i w razie czego odpowiednio zmienić wynik.

    } // koniec if

    System.out.println(wynik); ← Wyświetlenie wyniku („pułdo”, chyba że domyślny wynik zostanie zmieniony na „trafiony” lub „zatopiony”).

    return wynik; ← Zwrócenie wyniku do metody wywołującej.

} // koniec metody
```

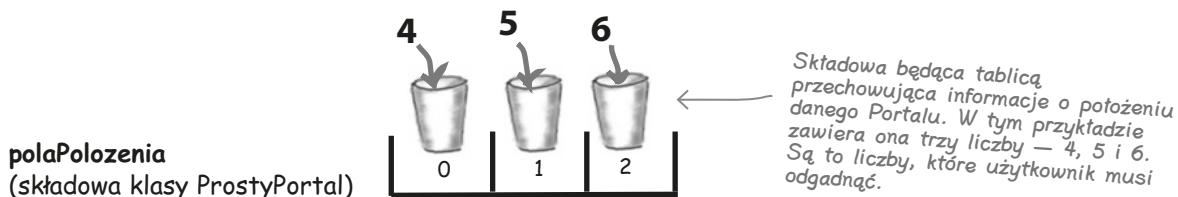
Jak poprawimy błąd?

Musimy mieć sposób, aby wiedzieć, czy dana komórka została już wcześniej trafiona. Przeanalizujemy różne możliwości, jednak najpierw przypomnimy sobie to, co już wiemy...

Dysponujemy wirtualnym wierszem zawierającym 7 komórek, a Portal zajmuje trzy kolejne komórki tego wiersza. Na poniższym rysunku przedstawiony został wirtualny wiersz oraz Portal umieszczony w komórkach o indeksach 4, 5 i 6.



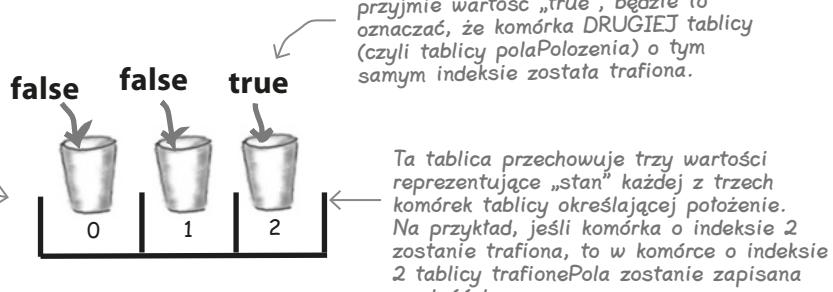
Obiekt ProstyPortal posiada składową — tablicę typu int — określającą zajmowane przez niego komórki.



1 Opcja pierwsza

Można by utworzyć drugą tablicę i za każdym razem, gdy użytkownik trafi jedną z komórek zajmowanych przez Portal, zapisywać to trafienie w drugiej tablicy, a następnie, przy każdym kolejnym trafieniu, sprawdzać, czy komórka nie została „trafiona” wcześniej.

Tablica trafionePola
(to mogłyby być tablica typu boolean, zdefiniowana jako składowa klasy Portal.)



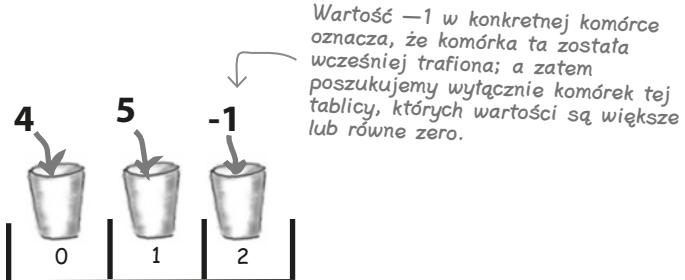
Pierwsza opcja jest zbyt niezgrabna

Wydaje się, że pierwsze rozwiązanie jest bardziej pracomilne niż moglibyśmy się tego spodziewać. Wymaga ono, aby za każdym razem, gdy użytkownik trafi w komórkę zajmowaną przez Portal, zmienić stan drugiej tablicy (`trafionePola`) — wcześniej jednak i tak trzeba przeanalizować tablicę `trafionePola`, aby sprawdzić, czy komórka nie została trafiona. Metoda ta będzie działać, jednak musi być jakieś lepsze rozwiązanie.

② Opcja druga

Można by korzystać tylko z jednej — oryginalnej — tablicy i zmieniać zawartość każdej „trafionej” komórki na -1 . Dzięki temu musimy sprawdzać i modyfikować tylko JEDNĄ tablicę.

`polaPolozenia`
(składówka klasy
`ProstyPortal`)



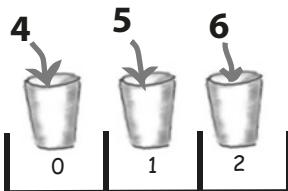
Opcja druga jest lepsza, lecz wciąż nieco niezgrabna

Opcja druga jest nieco mniej niezgrabna od opcji pierwszej, jednak wciąż nie jest zbyt efektywna. Nadal trzeba by przeglądać wszystkie trzy komórki (wartości indeksu), nawet jeśli jedna lub więcej z nich zostało „trafionych” (i zawierają wartość -1). Wciąż musi istnieć jakieś lepsze rozwiązanie.

3 Opcja trzecia

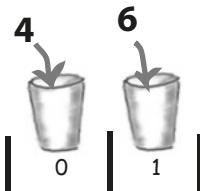
Usuwamy każdą komórkę tablicy odpowiadającą trafionemu polu, a następnie zmniejszamy tablicę. Problem polega na tym, że nie można zmieniać wielkości tablicy, a zatem musimy utworzyć nową tablicę i skopiować do niej pozostałe elementy tablicy oryginalnej.

tablica polaPolozenia
(ZANIM jedna
z komórek została
trafiona)



Początkowo tablica zawiera trzy komórki i wszystkie te komórki przeglądamy, porównując wartość podaną przez gracza z zawartością komórki (4, 5 i tak dalej).

tablica polaPolozenia
PO trafieniu komórki
„5” o indeksie 1



W momencie trafienia komórki „5” tworzymy nową, mniejszą tablicę zawierającą wszystkie komórki tablicy oryginalnej, za wyjątkiem komórki trafionej, następnie zapisujemy tę tablicę w zmiennej, w której wcześniej przechowywane było odwołanie do oryginalnej tablicy polaPolozenia.

To trzecie rozwiązywanie byłoby jeszcze lepsze, gdyby używana tablica mogła się zmniejszać, dzięki czemu nie musielibyśmy tworzyć nowej, mniejszej tablicy, kopiować do niej pozostałych komórek tablicy oryginalnej i zapamiętywać odwołania do nowej tablicy.

**Oryginalny kod przygotowawczy fragmentu
metody sprawdz():**

POWTÓRZ dla każdej z komórek tablicy typu int →
// PORÓWNAJ pole wskazane przez gracza z polami położenia
JEŚLI użytkownik trafił

INKREMENTUJ ilość trafień →
// SPRAWDŹ, czy to była ostatnia komórka Portalu

JEŚLI ilość trafień jest równa 3, ZWRÓĆ wynik „zatopiony” →
W PRZECIWNYM RAZIE, jeśli Portal nie został zatopiony,
ZWRÓĆ wynik „trafiony”

KONIEC JEŚLI

W PRZECIWNYM RAZIE pole wskazane przez użytkownika nie jest zajmowane przez Portal, ZWRÓĆ wynik „pułdo”

KONIEC JEŚLI

KONIEC POWTÓRZ

Życie byłoby znacznie prostsze, gdybyśmy tylko mogli zmienić ten kod na następujący:

POWTÓRZ dla każdej z pozostałej komórki tablicy typu int →
// PORÓWNAJ pole wskazane przez gracza z polami położenia
JEŚLI użytkownik trafił

USUŃ tę komórkę z tablicy →
// SPRAWDŹ, czy to była ostatnia komórka Portalu

JEŚLI tablica jest pusta, ZWRÓĆ wynik „zatopiony” →
W PRZECIWNYM RAZIE, jeśli Portal nie został zatopiony,
ZWRÓĆ wynik „trafiony”
KONIEC JEŚLI

W PRZECIWNYM RAZIE pole wskazane przez użytkownika nie jest zajmowane przez Portal, ZWRÓĆ wynik „pułdo”
KONIEC JEŚLI

KONIEC POWTÓRZ



Gdybym tylko mogła znaleźć tablicę,
która w przypadku usunięcia elementu
zmniejsza się. I taką, której nie trzeba
przeglądać w całości, lecz wystarczy
zapytać, czy zawiera poszukiwany
element. I taką, która pozwalałaby na
pobieranie zawartości bez znajomości
indeksu komórki. To byłoby wspaniałe.
Ale wiem, że to jedynie fantazja...

Obudź się i przejrzyj bibliotekę

Może to magia, ale faktycznie istnieje taka „tablica”.

Jednak nie jest to normalna tablica — to klasa **ArrayList**.

Klasa należąca do podstawowej biblioteki Javy (API).

Java Standard Edition (czyli ta wersja Javy, na której pracujesz, no chyba że używasz wersji Micro Edition przeznaczonej dla niewielkich urządzeń, ale wierz mi, *wiedziałbyś o tym*) jest dostarczana wraz z setkami gotowych do użycia klas przypominających nieco nasze klasy oznaczane w książce jako  Gotowy kod, lecz różniące się od nich tym, że już są skompilowane.

A to oznacza, że nie trzeba ich ręcznie wpisywać w programie.

Wystarczy ich użyć.

Jedna z gazylionów klas tworzących bibliotekę Javy.
Możesz ich używać w swoich programach, tak jakbyś sam je napisał.

ArrayList

add(Object elem) Dodaje przekazany obiekt do listy.	remove(int index) Usuwa z listy obiekt określony przez parametr index.
remove(Object elem) Usuwa z listy przekazany obiekt (jeśli się w niej znajduje).	contains(Object elem) Zwraca wartość true, jeśli przekazany obiekt odpowiada jednemu z obiektów znajdujących się na liście.
isEmpty() Zwraca wartość true, jeśli lista nie zawiera żadnych elementów.	indexOf(Object elem) Zwraca indeks przekazanego obiektu lub wartość -1, jeśli go nie ma na liście.
size() Zwraca liczbę elementów aktualnie znajdujących się na liście.	get(int index) Zwraca obiekt, który aktualnie jest zapisany w elemencie listy o podanym indeksie.

(Uwaga: metoda `add(Object elem)` wygląda nieco bardziej dziwnie niż ta, którą przedstawiliśmy na tej stronie... jej prawdziwą postać przedstawimy w dalszej części książki. Jak na razie możesz ją sobie jednak wyobrażać właśnie w takiej postaci — jako metodę pobierającą obiekt, który chcesz dodać do listy).

To jedynie przykłady NIEKTÓRYCH, wybranych metod klasy `ArrayList`.

Niektóre możliwości klasy `ArrayList`

Nie przejmuj się tą nową składnią z klasą <Jajko>
umieszczoną w nawiasach kątowych; oznacza
mniej więcej tyle: „utwórz listę obiektów klasy
Jajko”.

1 Tworzenie tablicy.

```
ArrayList<Jajko> mojaTbl = new ArrayList<Jajko>();
```

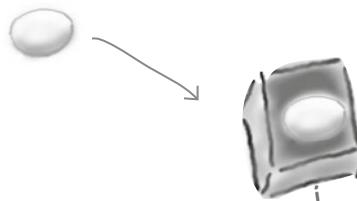


Utworzony został nowy obiekt
`ArrayList`, początkowo jest bardzo
mały, gdyż jest pusty.

2 Dodanie czegoś do tablicy

```
Jajko j = new Jajko();
```

```
mojaTbl.add(j);
```

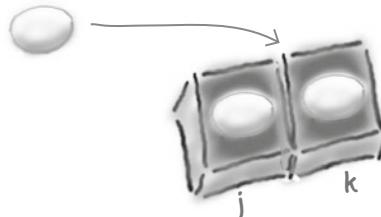


Teraz obiekt `ArrayList` staje się
„pułapkiem”, w którym zostaje
umieszczony obiekt `Jajko`.

3 Ponowne dodanie czegoś do tablicy.

```
Jajko k = new Jajko();
```

```
mojaTbl.add(k);
```



Obiekt `ArrayList` ponownie jest
powiększony, aby można w nim
było umieścić drugi obiekt `Jajko`.

4 Określenie ilości elementów w tablicy.

```
int ilosc = mojaTbl.size();
```

Obiekt `ArrayList` zawiera 2 obiekty,
zatem metoda `size()` zwraca wartość 2.

5 Sprawdzenie, czy tablica zawiera pewien obiekt.

```
boolean czyJest = mojaTbl.contains(j);
```

`ArrayList` faktycznie ZAWIERA
obiekt `Jajko`, do którego odwołuje się
zmienna „`j`”, zatem metoda `contains()`
zwraca wartość true.

6 Określenie położenia obiektu (czyli jego indeksu).

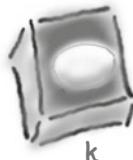
```
int indeks = mojaTbl.indexOf(k);
```

Pierwszy element tablicy `ArrayList` zawsze
ma indeks 0, a ponieważ obiekt, do którego
odwołuje się zmienna „`k`”, był drugim
obiektem umieszczonym w tablicy, zatem
metoda `indexOf()` zwraca wartość 1.

7 Sprawdzenie, czy tablica jest pusta.

```
boolean czyPusta = mojaTbl.isEmpty();
```

Bez wątpienia tablica NIE jest pusta,
zatem metoda `isEmpty()` zwraca
wartość false.



Hej, popatrz – tablica się
zmniejszyła!

8 Usunięcie czegoś z tablicy.

```
mojaTbl.remove(j);
```



Zaostrz ołówek

Wypełnij puste miejsca w poniższej tabeli. W tym celu przeanalizuj kod wykorzystujący obiekt ArrayList przedstawiony w lewej kolumnie i zapisz w prawej kolumnie kod wykorzystujący normalną tablicę, który, według Ciebie, miałyby takie same możliwości. Nie oczekujemy, że podany przez Ciebie kod będzie całkowicie poprawny, zatem po prostu postaraj się zrobić to jak najlepiej.

ArrayList

Zwyyczajna tablica

ArrayList<String> mojaTbl = new ArrayList<String>();	String[] mojaTbl = new String[2];
String a = new String("hooohooo"); mojaTbl.add(a);	String a = new String("hooohooo");
String b = new String("Żaba"); mojaTbl.add(b);	String b = new String("Żaba");
int ilosc = mojaTbl.size();	
String o = mojaTbl.get(1);	
mojaTbl.remove(1);	
boolean czyJest = mojaTbl.contains(b);	

Nie istnieja
grupie pytania

P: Zgoda, klasa **ArrayList** jest super, ale skąd miałem wiedzieć, że w ogóle istnieje?

O: Właściwie pytanie to powinno brzmieć: „Jak się dowiedzieć, czy ta klasa istnieje w API?” i stanowi ono kluczowy czynnik niezbędny do odniesienia sukcesu w roli programisty używającego Javy; nie wspominając w ogóle o możliwości bycia leniwym przy jednoczesnym zachowaniu szans na tworzenie programów. Zapewne zaskoczy Cię, jak wiele czasu można zaoszczędzić, kiedy ktoś inny wykonał całą brudną robotę, zostawiając Ci jedynie napisanie tych najfajniejszych fragmentów.

Ale oddaliśmy się nieco od właściwego tematu pytania... Odpowiadając na nie krótko, należałoby stwierdzić, że powinieneś poświęcić nieco czasu na naukę i poznanie Java API. Nieco dłuższą odpowiedź znajdziesz pod koniec tego rozdziału, gdzie opisaliśmy, jak możesz poznać API.

P: Ale to jest naprawdę trudne zadanie. Nie tylko muszę wiedzieć, że biblioteka Javy zawiera klasę **ArrayList**, lecz, co ważniejsze, że jest to klasa, która potrafi robić to, na czym mi zależy! Zatem w jaki sposób można przejść od problemu „muszę-to-zrobić” do „rozwiążanie-wykorzystujące-Java API”?

O: Teraz trafiliś w sedno. Jednak w chwili, kiedy zakończysz lekturę tej książki, będziesz dysponować dobrą znajomością języka, a dalsza część procesu nauki będzie polegać właśnie na określeniu, jak dotrzeć do rozwiązania problemu, tworząc przy tym jak najmniej własnego kodu. Jeśli tylko jesteś w stanie, to znajdź w sobie na tyle cierpliwości, aby przeczytać kilka kolejnych stron. Pod koniec tego rozdziału poruszmy zagadnienia, które będą zawierać odpowiedzi na Twoje pytania.



Java bez tajemnic

W tym tygodniu przeprowadzimy wywiad z:
Obiektem ArrayList na temat tablic

Autorzy: A zatem jesteś podobny do tablic, nieprawdaż?

ArrayList: Chyba w ich marzeniach! *Ja*, na szczęście, jestem obiektem.

Autorzy: Jeśli się nie mylimy, to tablice też są obiektami. Istnieją na stercie razem ze wszystkimi innymi obiektami w programie.

ArrayList: Jasne, oczywiście że tablice istnieją na stercie, jednak i tak mogą sobie tylko pomarzyć, żeby być takie jak ja. Pożerki. Obiekty mają stan i działanie. Prawda? Co do tego nie ma chyba wątpliwości. Ale czy kiedykolwiek spróbowałeś wywołać jakąś metodę tablicy?

Autorzy: W zasadzie, kiedy o tym wspomniałeś, musimy przyznać, że nie. Ale w ogóle jaką metodę tablicy mielibyśmy wywoływać? Obchodziło nas jedynie wywoływanie metod obiektów, które były przechowywane w tablicy, a nie samej tablicy. Poza tym, zapisując dane w tablicy i pobierając je z niej, możemy się posługiwać specjalną składnią.

ArrayList: Czyżby? Chcacie mi powiedzieć, że udało wam się usunąć coś z tablicy? (Rany! Gdzie wyście się uczyli? W Mc'Javie?)

Autorzy: Oczywiście, że mogę coś pobrać z tablicy. Na przykład piszę: `Pies p = tablicaPsów[1]` i pobieram z tablicy obiekt `Pies` zapisany w komórce o indeksie 1.

ArrayList: No dobrze, będę mówić wolno i wyraźnie, żebyście mogli zrozumieć. W ten sposób *nie* usuwaliście, powtarzam — *nie* usuwaliście obiektu z tablicy. Skopiowaliście jedynie *odwołanie do obiektu `Pies`* i zapisali je w innej zmiennej.

Autorzy: Acha... Teraz rozumiemy, co masz na myśli. Faktycznie, nie usuwaliśmy obiektu `Pies` z tablicy. Ciągle tam jest. Ale z tego co wiemy, wystarczy przypisać odpowiedniemu odwołaniu wartość `null`, aby usunąć obiekt.

ArrayList: Ale ja jestem doskonałym obiektem — dysponuję metodami i potrafię robić różne rzeczy, takie jak faktyczne usuwanie odwołań do obiektów z mojej listy, a nie jedynie przypisywanie im wartości `null`. I potrafię *dynamicznie* zmieniać wielkość mojej listy (możecie to sprawdzić). Spróbujcie zrobić to samo z *tablicą*!

Autorzy: Hm... Nie znamy wspominać o tym, ale krążą plotki, że nie jesteś niczym więcej niż sławną, lecz nieco mniej efektywną tablicą. Że w rzeczywistości jesteś jedynie otoczką wokół normalnej tablicy i dodajesz do niej tylko kilka metod, które inaczej mielibyśmy napisać sami, takich jak zmiana wielkości. I, skoro już o tym mówimy, to nawet nie jesteś w stanie przechowywać wartości typów podstawowych! Czyż to nie jest poważne ograniczenie?

ArrayList: Po prostu nie wierzę, że kupiliście tę bajeczkę. Nie, *nie* jestem jedynie nieco mniej efektywną tablicą. Przyznaję, że jest kilka *niewykle* rzadkich sytuacji, w których normalne tablice mogą być minimalnie, powtarzam, *minimalnie* szybsze ode mnie. Jednak wyrzeczenie się całej mojej *potęgi* nie jest warte tego *malutkiego* wzrostu efektywności. Cała ta *elastyczność*. A jeśli chodzi o typy podstawowe, to oczywiście, że można umieszczać je we mnie, o ile tylko zostaną umieszczone w obiekcie — jak w „opakowaniu” (dowiecie się o nich więcej w rozdziale 10.). Natomiast w wersji 5.0 Javy to „opakowywanie” (i „rozpakowywanie” podczas odczytywania wartości ze mnie) jest realizowane automatycznie. No dobrze, *potwierdzam*, że w przypadku operowania na wartościach typów podstawowych tablice faktycznie mogą być nieco szybsze ze względu na to całe „opakowywanie” i „rozpakowywanie”, ale... kto w *naszych czasach* korzysta jeszcze z wartości typów podstawowych?

Rany, patrzcie, która godzina! *Jestem już spóźniony na Impreżkę*. Ale kiedyś będziemy musieli pogadać jeszcze raz.

Porównanie obiektu **ArrayList** i normalnej tablicy

ArrayList	Zwyyczajna tablica
ArrayList<String> mojaTbl = new ArrayList<String>();	String[] mojaTbl = new String[2];
String a = new String("hooohooo"); mojaTbl.add(a);	String a = new String("hooohooo"); mojaTbl[0] = a;
String b = new String("żaba"); mojaTbl.add(b);	String b = new String("żaba"); mojaTbl[1] = b;
int ilosc = mojaTbl.size();	int ilosc = mojaTbl.length;
String o = mojaTbl.get(1);	String o = mojaTbl[1];
mojaTbl.remove(1);	mojaTbl[1] = null;
boolean czyJest = mojaTbl.contains(b);	boolean czyJest = false; for (String elem : mojaTbl) { if (b.equals(mojaTbl[i])) { czyJest = true; break; } }

Dopiero tutaj widać prawdziwą różnicę...

Zwróć uwagę, że w przypadku stosowania **ArrayList** wykorzystujesz obiekt, a zatem wywołujesz stare dobre metody starego dobrego obiektu, posługując się przy tym starym dobrym operatorem kropki.

W przypadku wykorzystania *tablicy* używasz *specjalnej składni* (takiej jak `tablica[0] = coś tam`), która nie jest stosowana w żadnych innych okolicznościach. Pomimo faktu, że tablica jest obiektem, to obiekt ten żyje w swoim własnym, specjalnym świecie i nie można wywoływać jego metod; istnieje jednak możliwość korzystania z jednej składowej tego obiektu — `length`.

Porównanie klasy `ArrayList` ze zwyczajną tablicą

1 Stara, dobra, normalna tablica musi znać swoją wielkość w momencie tworzenia.

Natomiast w przypadku obiektów `ArrayList` po prostu tworzymy obiekt tego typu. Nigdy nie musimy określić jego wielkości, gdyż obiekty te odpowiednio się powiększą lub zmniejszą wraz z dodawaniem lub usuwaniem ich zawartości.

```
new String[2]; ← Potrzebna jest wielkość tablicy.
```

```
new ArrayList<String>()
```

Określenie wielkości nie jest potrzebne (choć jeśli chcemy, to możemy ją podać).

2 Aby umieścić obiekt w normalnej tablicy, trzeba zapisać go w jej konkretnym miejscu.

```
mojaTbl[1] = b;
```

↑ Potrzebny jest indeks.

Jeśli indeks znajduje się poza zakresem tablicy (na przykład w sytuacji, gdy zadeklarowano tablicę o wielkości 2, a próbujesz zapisać coś w komórce o indeksie 3), to w trakcie działania programu zostanie zgłoszony błąd.

W przypadku obiektu `ArrayList` możesz określić indeks, używając metodę `add` o postaci: `add(liczbaInt, elemObject);`, lub ograniczyć się do wywołania `add(elemObject)`, a obiekt `ArrayList` w razie konieczności powiększy się, robiąc miejsce na kolejne elementy.

```
mojaTbl.add(b);
```

↑ Bez indeksu.

3 Tablice używają specjalnej składni, która nie jest wykorzystywana w żadnych innych konstrukcjach języka.

Natomiast obiekty `ArrayList` to zwyczajne obiekty, zatem można się nimi posługiwać bez konieczności korzystania z jakiegokolwiek specjalnej składni.

```
mojaTbl[1]
```

↑ Nawiasy kwadratowe to specjalna składnia używana wyłącznie podczas operacji na tablicach.

4 W wersji Java 5.0 obiekty `ArrayList` są parametryzowane.

Napisaliśmy przed chwilą, że w odróżnieniu od tablic obiekty `ArrayList` nie wymagają stosowania żadnej specjalnej składni. Jednak w rzeczywistości *używają* czegoś specjalnego, dodatku wprowadzonego w Javie 5.0 (Tiger) — *sparametryzowanych typów*.

```
ArrayList<String>
```

↑ Typ `<String>` zapisany w nawiasach kątowych jest tak zwany parametrem typu. Konstrukcja `ArrayList<String>` oznacza „lista obiektów typu `String`”; taka lista będzie czymś innym niż `ArrayList<Pies>`, czyli „lista obiektów typu `Pies`”.

Przed pojawiением się wersji 5.0 Javy nie było możliwości określenia *typu* elementów umieszczanych na liście `ArrayList`, a zatem, z punktu widzenia kompilatora, wszystkie obiekty `ArrayList` były niejednorodnymi kolekcjami obiektów. Jednak obecnie, dzięki składni `<tuUmieśćTyp>`, możemy deklarować i tworzyć obiekty `ArrayList`, które znają typ swojej zawartości (i dbają o to, by umieszczane w nich obiekty faktycznie były tego typu). Typom sparametryzowanym wykorzystywanym w obiektach `ArrayList` przyjrzymy się bardziej szczegółowo w rozdziale poświęconym kolekcjom, a zatem na razie nie musisz sobie zwracać głowy tą specyficzną składnią z nawiasami kątowymi (`<>`) używaną przy tworzeniu obiektów `ArrayList`. Wystarczy, byś zapamiętał, że jest to specjalna konstrukcja, dzięki której kompilator będzie pozwalał na umieszczanie na liście `ArrayList` jedynie obiektów określonego typu (*podanego w nawiasach kątowych*).

kod przygotowawczy

kod testowy

kod właściwy

Poprawmy kod klasy Portal

Pamiętasz zapewne, że tak wygląda kod klasy, w której znaleźliśmy błąd:

```
class Portal {  
    int[] polaPolozenia;  
    int iloscTrafien;  
  
    public void setPolaPolozenia(int[] ppol) {  
        polaPolozenia = ppol;  
    }  
  
    public String sprawdz(String stringPole) {  
        int strzał = Integer.parseInt(stringPole);  
        String wynik = "pudło";  
  
        for (int pole : polaPolozenia) {  
            if (pole == strzał) {  
                wynik = "trafiony";  
                iloscTrafien++;  
  
                break;  
            }  
        } // koniec pętli  
  
        if (iloscTrafien == polaPolozenia.length) {  
            wynik = "zatopiony";  
        }  
        System.out.println(wynik);  
        return wynik;  
    } // koniec metody  
} // koniec klasy
```

Zmieniliśmy nazwę klasy z ProstyPortal na Portal, gdyż będziemy teraz zajmować się zaawansowaną wersją naszej gry. Jednak jest to ten sam kod, który był przedstawiony w poprzednim rozdziale.

To właśnie tutaj występuje błąd. Każde pole podane przez gracza i zajmowane przez portal traktowaliśmy jako trafienie bez sprawdzania, czy dana komórka została wcześniej trafiona czy nie.

Nowa i poprawiona klasa Portal

```

import java.util.ArrayList; ←
Na razie zignoruj ten
wiersz kodu; wróćmy
do niego pod koniec
rozdziatu.

class Portal {
    private ArrayList<String> polaPolozenia;
    // int iloscTrafien; (już niepotrzebne)
    // Zmieniamy tablicę int[]
    // na tablicę ArrayList.

    public void setPolaPolozenia(ArrayList<String> ppol) {
        polaPolozenia = ppol;
    }

    public String sprawdz(String ruch) {
        String wynik = "pudło";
        Nowa, ulepszona nazwa argumentu.

        int indeks = polaPolozenia.indexOf(ruch);
        Sprawdzamy, czy pole wskazane przez gracza
        znajduje się w tablicy ArrayList. W tym celu
        prosimy o podanie indeksu pola. Jeśli pola nie
        będzie, to metoda indexOf() zwróci wartość -1.

        if (indeks >= 0) { ←
            Jeśli indeks ma wartość większą lub
            równą zero, to wskazane przez gracza
            pole na pewno znajduje się w tablicy.
            W takim razie usuwamy je.

            polaPolozenia.remove(indeks); ←

            if (polaPolozenia.isEmpty()) { ←
                Jeśli tablica jest pusta,
                to było to ostateczne
                trafienie.

                wynik = "zatopiony";
            } else {
                wynik = "trafiony";
            } // koniec if
        } // koniec if

        return wynik;
    } // koniec metody
} // koniec klasy

```



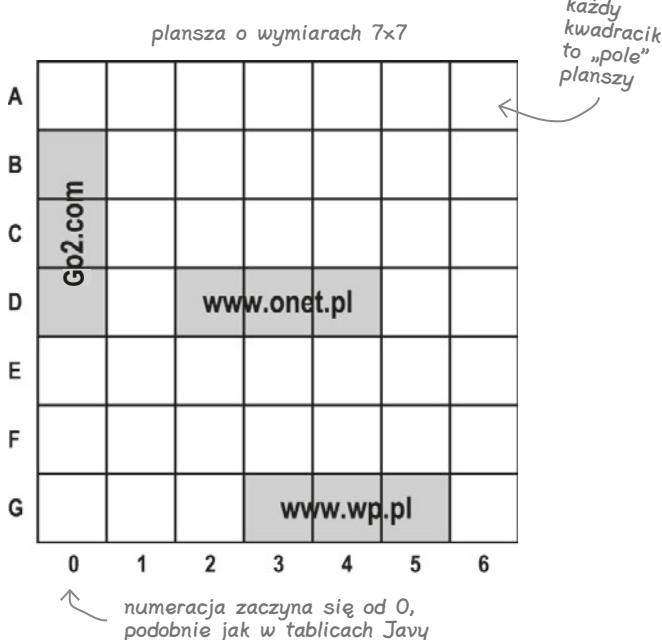
Napiszmy właściwą wersję gry „Zatopić portal”

Do tej pory zajmowaliśmy się uproszczoną wersją naszej gry; nadszedł jednak czas, aby napisać wersję ostateczną. Zamiast wiersza użyjemy kwadratowej planszy. A zamiast jednego portalu na planszy umieścimy trzy.

Cel: Zatopić wszystkie Portale komputera w jak najmniejszej ilości ruchów. Gracz otrzymuje ocenę na podstawie uzyskanych wyników.

Przygotowanie: W momencie uruchamiania gry komputer umieszcza trzy Portale na **wirtualnej planszy o wymiarach 7×7** . Po wykonaniu tej operacji program prosi gracza o wykonanie pierwszego ruchu.

Sposób prowadzenia gry: Nie poznaliśmy jeszcze sposobów tworzenia graficznego interfejsu użytkownika, a zatem ta wersja gry będzie obsługiwana z poziomu wiersza poleceń. Komputer będzie prosił o podanie ruchu (współrzędnych pola), które powinieneś wpisać w wierszu poleceń (w postaci: „A3”, „C5” itp.). W odpowiedzi na podane współrzędne na ekranie zostanie wyświetlony komunikat: „trafiony”, „pułdu” lub „zatopiłeś Onet.pl” (przy czym podana nazwa może być inna, zależnie od tego, jaki portal tym razem miał pecha). Kiedy wszystkie trzy portale zostaną zatopione i zostanie po nich jedynie rozwiewany wiatrem obłoczek w kształcie liczby 404, gra zakończy się, a program wyświetli ocenę gracza.



Masz za zadanie napisać grę „Zatopić portal”, przy czym każdy z zatapianych portali zajmuje dokładnie trzy pola planszy.

Fragment przebiegu gry

```
Wiersz poleceń
T:\>java PortalGraMax
Twoim celem jest zatopienie trzech portali.
onet.pl, wp.pl, Go2.com
Postaraj się je zatopić w jak najmniejszej ilości ruchów.
Podaj pole: d1
pułdu
Podaj pole: e2
pułdu
Podaj pole: f0
trafiony
Podaj pole: f1
trafiony
Podaj pole: f2
HUC! Zatopiłeś portal onet.pl :(
zatopiony
Podaj pole: b4
pułdu
Podaj pole: c3
trafiony
Podaj pole: b3
trafiony
Podaj pole: a3
HUC! Zatopiłeś portal wp.com :(
zatopiony
```

Jakie zmiany należy wprowadzić?

Należy zmienić trzy klasy: Portal (która aktualnie będzie nosić właśnie tę nazwę zamiast *ProstyPortal*), klasę gry (*PortalGraMax*) oraz klasę pomocniczą gry (która na razie nie będziemy się przejmować).

A Klasa Portal

◎ Dodanie zmiennej *nazwa*

służącej do przechowywania nazwy Portalu (na przykład: „onet.pl”), tak aby można wyświetlić jego nazwę, gdy Portal zostanie zatopiony (patrz przykładowy przebieg gry przedstawiony na poprzedniej stronie).

B Klasa PortalGraMax

◎ Tworzenie *trzech* obiektów Portal zamiast jednego.

◎ Nadanie każdemu z obiektów Portal *nazwy*.

Wywołanie metody ustawiającej dla każdego obiektu Portal, tak aby każdemu z nich nadać nazwę i zapisać ją w odpowiedniej składowej.

Dalszy ciąg modyfikacji klasy PortalGraMax...

- ◎ Umieszczenie wszystkich trzech Portali na dwuwymiarowej planszy, a nie w pojedynczym wierszu.

Jeśli poszczególne Portale mają być rozmieszczane na planszy losowo, to czynność ta będzie bardziej złożona niż była w uproszczonej wersji programu. Nie mamy zamieru zajmować się tu obliczeniami matematycznymi, dlatego też algorytm określający położenie Portali na planszy został umieszczony w klasie *PomocnikGry* (patrz Kod gotowy do użycia).

- ◎ Sprawdzanie *wszystkich* trzech Portali po wykonaniu ruchu przez gracza, a nie tylko jednego, jak było wcześniej.

- ◎ Kontynuacja gry (to znaczy odczytywanie ruchów gracza i sprawdzanie pozostałych Portali) *aż do momentu, gdy wszystkie Portale zostaną „zatopione”*.

- ◎ Usunięcie kodu z metody *main()*. Kod zarządzający grą w jej prostej wersji był umieszczony wewnątrz metody *main()* po to, aby... była ona prosta. Jednak nie chcemy, aby pełna wersja gry była napisana w taki sposób.

3 klasy:

używany do pobierania ruchów gracza i określania położenia Portali
tworzy i prowadzi grę przy użyciu

PortalGraMax
Klasa gry. Tworzy obiekty Portal, pobiera dane wprowadzane przez gracza i prowadzi grę aż do momentu, gdy wszystkie Portale zostaną zatopione.

Portal
Obiekty Portal. Obiekty Portal znajdują swoją nazwę, położenie i wiedzą, jak sprawdzić, czy użytkownik trafił w zajmowane przez nie pole.

PomocnikGry
Klasa pomocnicza (Kod gotowy do użycia). Wie, jak pobierać dane wprowadzane przez gracza w wierszu poleceń i określać położenia Portali.

5 obiektów:



PortalGraMax



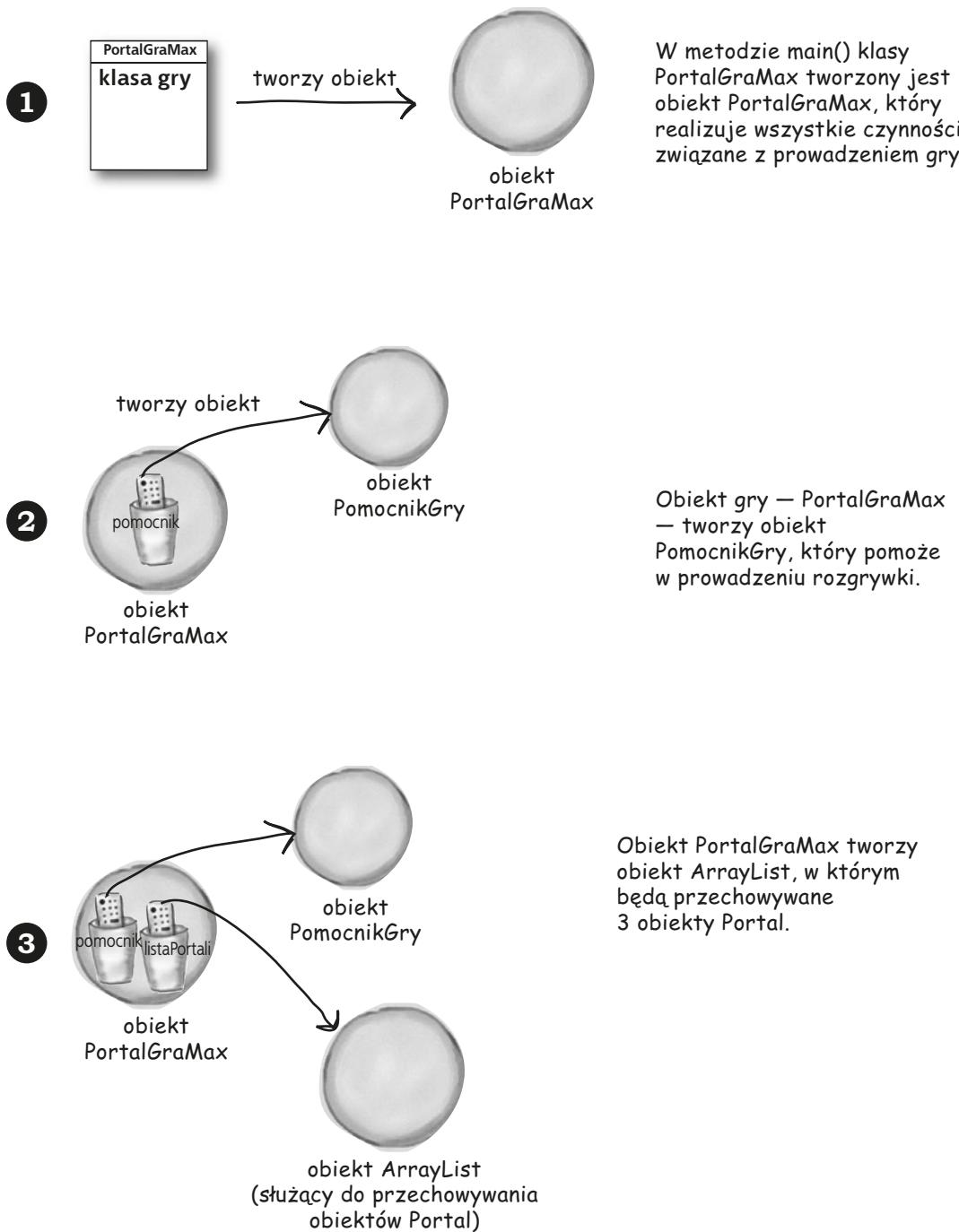
Por.
Por.
Portal



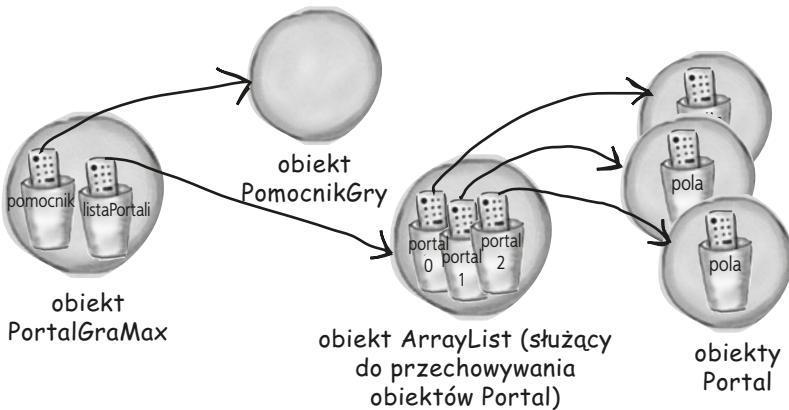
PomocnikGry

Oraz 4 obiekty *ArrayList*: jeden używany w obiekcie *PortalGraMax* oraz po jednym w każdym z trzech obiektów *Portal*.

Co (i kiedy) robią poszczególne obiekty w grze?



4

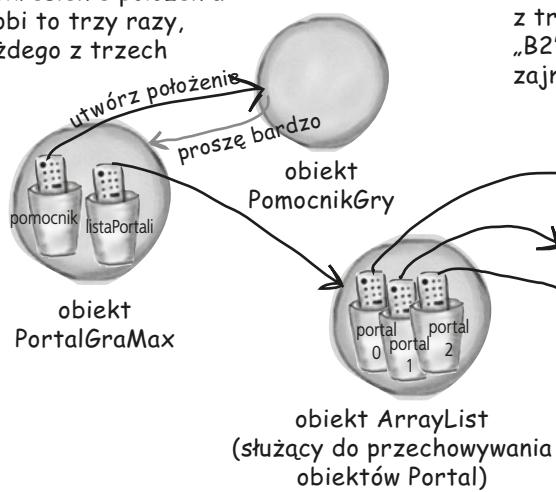


Obiekt PortalGraMax tworzy trzy obiekty Portal (i umieszcza je na liście ArrayList).

Obiekt PortalGraMax prosi obiekt pomocnika gry o określenie położenia obiektu Portal (robi to trzy razy, po jednym dla każdego z trzech obiektów Portal).

Obiekt PortalGraMax nadaje położenie (określone i zwrócone przez obiekt pomocnika gry) każdemu z trzech obiektów Portal, przykładowo: „A2”, „B2” itd. Każdy z obiektów Portal umieszcza trzy zajmowane przez niego pola na liście ArrayList.

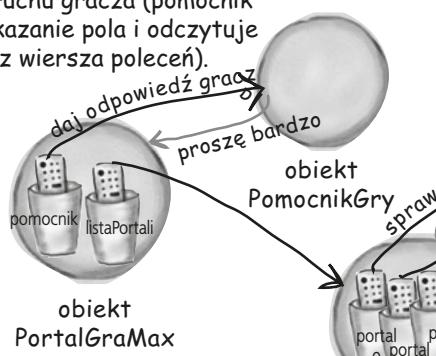
5



Obiekt PortalGraMax prosi pomocnika gry o odczytanie ruchu gracza (pomocnik prosi gracza o wskazanie pola i odczytuje jego współrzędne z wiersza poleceń).

Obiekt PortalGraMax przegląda listę wszystkich obiektów Portal i prosi każdy z nich o sprawdzenie ruchu wykonanego przez gracza. Każdy z obiektów Portal sprawdza wskazane pole i zwraca wynik („trafiony”, „pułło” itd.).

6



I tak toczy się gra... sprowadza się do pobierania ruchu wykonanego przez gracza, sprawdzenie tego ruchu przez każdy z obiektów Portal i oczekiwania aż wszystkie Portale zostaną zatopione.

Klasa PortalGraMax (gra)

kod przygotowawczy

kod testowy

kod właściwy

PortalGraMax
PomocnikGry pomocnik ArrayList listaPortali int iloscRuchow
przygotujGre() rozpoczniGre() sprawdzRuchGracza() zakonczGre()

Kod przygotowawczy właściwej klasy PortalGraMax

Klasa PortalGraMax ma trzy podstawowe zadania: przygotować rozgrywkę, prowadzić rozgrywkę aż do momentu zatopienia wszystkich Portali i zakończyć rozgrywkę. Choć można by realizować te zadania w trzech metodach, to jednak drugie z nich (prowadzenie rozgrywki) podzieliliśmy na *dwie* metody, tak aby każda z nich była prostsza. Mniejsze i prostsze metody (co jednocześnie oznacza, że mają one mniejsze możliwości funkcjonalne) ułatwiają testowanie, usuwanie błędów i wprowadzanie modyfikacji.

ZADEKLARUJ i utwórz składową typu *PomocnikGry* o nazwie *pomocnik*.

ZADEKLARUJ i utwórz składową typu *ArrayList* o nazwie *listaPortali*, w której będzie przechowywana lista Portali (początkowo będą tworzone trzy Portale).

ZADEKLARUJ zmienną typu int służącą od przechowywania ilości ruchów wykonanych przez gracza (abyśmy pod koniec rozgrywki mogli podać uzyskany wynik). Nadaj zmiennej nazwę *iloscruchow* i przypisz jej początkową wartość 0.

ZADEKLARUJ metodę *przygotujGre()* służącą do utworzenia obiektów Portal oraz określenia ich nazw i położenia na planszy. Wyświetl krótkie instrukcje dla gracza.

ZADEKLARUJ metodę *rozpoczniGre()*, która prosi gracza o podanie ruchu i wywołuje metodę *sprawdzRuchGracza()* aż do momentu, gdy wszystkie Portale zostaną usunięte z gry.

ZADEKLARUJ metodę *sprawdzRuchGracza()*, która wywołuje metodę *sprawdz()* wszystkich pozostających w grze Portali.

ZADEKLARUJ metodę *zakonczGre()*, która, na podstawie ilości wykonanych ruchów, wyświetla komunikat o wynikach uzyskanych przez gracza.

METODA: void przygotujGre()

// utwórz trzy obiekty Portal i określ ich nazwy

UTWÓRZ trzy obiekty Portal.

OKREŚL nazwę każdego z obiektów Portal.

DODAJ wszystkie obiekty Portal do tablicy *listaPortali* (typu ArrayList).

POWTÓRZ dla każdego z obiektów Portal zapisanych w tablicy *listaPortali*.

WYWŁAŻ metodę *rozmiescPortal()* obiektu pomocnika gry, aby pobrać losowo wybrane położenie Portalu (trzy komórki sąsiadujące ze sobą w pionie lub poziomie umieszczone na planszy o wymiarach 7x7).

OKREŚL położenie każdego z Portali na podstawie wyników zwróconych przez metodę *rozmiescPortal()*.

KONIEC POWTÓRZ

KONIEC METODY

ciąg dalszy implementacji metod:

METODA: *void rozpoczęjGre()*

POWTARZAJ, dopóki istnieje jakiś obiekt Portal.

POBIERZ ruch gracza poprzez wywołanie metody pomocnika gry *pobierzDaneWejsciowe()*.

PRZETWÓRZ ruch, posługując się metodą *sprawdzRuchGracza()*.

KONIEC POWTARZAJ

KONIEC METODY

METODA: *void sprawdzRuchGracza(String ruch)*

// sprawdzamy, czy któryś z Portali został trafiony (a może też zatopiony).

INKREMENTUJ liczbę ruchów przechowywanych w składowej *iloscruchow*.

PRZYPISZ zmiennej lokalnej wynik (typu *String*) wartość „pułło”, zakładając, że gracz chybi.

POWTÓRZ dla każdego Portalu w tablicy *listaportali*.

PRZETWÓRZ pole wskazane przez gracza, wywołując metodę *sprawdz()* obiektu Portal.

PRZYPISZ zmiennej wartość „trafiony” lub „zatopiony”, jeśli to będzie konieczne.

JEŚLI wynik ma wartość „zatopiony”, **TO usuń** Portal z tablicy *listaportali*.

KONIEC POWTÓRZ

WYŚWIETL wynik na ekranie.

KONIEC METODY

METODA: *void zakonczGre()*

WYŚWIETL ogólny komunikat o zakończeniu gry, a następnie:

JEŚLI liczba ruchów wykonanych przez gracza nie jest duża,

WYŚWIETL komunikat z gratulacjami.

W PRZECIWNYM RAZIE

WYŚWIETL obraźliwy komunikat.

KONIEC JEŻELI

KONIEC METODY



Zaostrz ołówek

W jaki sposób na podstawie kodu przygotowawczego można uzyskać właściwy kod klasy? W pierwszej kolejności tworzymy kod testujący, a następnie, bajt po bajcie tworzymy i testujemy metody. W dalszej części książki nie będziemy już przedstawiać kodu testującego, zatem już sam będziesz musiał przemyśleć i określić, co musisz wiedzieć, aby przetestować tworzone metody. A jakie metody należy przetestować i stworzyć w pierwszej kolejności? Sprawdź, czy jesteś w stanie napisać kod przygotowawczy dla niektórych testów. Na potrzeby tego ćwiczenia wystarczy, abyś napisał kod przygotowawczy lub nawet same zagadnienia kluczowe, jeśli jednak chcesz napisać prawdziwy kod testowy (w Javie), tym lepiej.

Kod klasy PortalGraMax (gra)

kod przygotowawczy kod testowy kod właściwy

```
import java.util.*;  
public class PortalGraMax {  
  
    ① { private PomocnikGry pomocnik = new PomocnikGry();  
        private ArrayList<Portal> listaPortali = new ArrayList<Portal>();  
        private int iloscRuchow = 0;  
  
        private void przygotujGre() {  
            // najpierw tworzymy portale i określamy ich położenie  
            Portal pierwszy = new Portal();  
            pierwszy.setNazwa("onet.pl");  
            Portal drugi = new Portal();  
            drugi.setNazwa("wp.com");  
            Portal trzeci = new Portal();  
            trzeci.setNazwa("Go2.com");  
            listaPortali.add(pierwszy);  
            listaPortali.add(drugi);  
            listaPortali.add(trzeci);  
  
            System.out.println("Twój celem jest zatopienie trzech portali.");  
            System.out.println("onet.pl, wp.pl, Go2.com");  
            System.out.println("Postaraj się je zatopić w jak najmniejszej ilości ruchów.");  
  
            for (Portal rozmieszczanyPortal : listaPortali) {  
                ArrayList<String> nowePolozenie = pomocnik.rozmiescPortal(3); ⑤  
                rozmieszczanyPortal.setPolaPolozenia(nowePolozenie);  
            } // koniec pętli for  
        } // koniec metody  
  
        private void rozpoczęjGre() { ⑦  
            while(!listaPortali.isEmpty()) {  
                String ruchGracza = pomocnik.pobierzDaneWejsciowe("Podaj pole:"); ⑧  
                sprawdzRuchGracza(ruchGracza); ⑨  
            } // koniec while  
            zakonczGre(); ⑩  
        } // koniec metody
```

Zaostrz ołówek

Sam opisz przedstawiony kod!

Dopasuj opisy podane u dołu każdej ze stron z liczbami zamieszczonymi w kodzie. Zapisz odpowiednie liczby w miejscach oznaczonych podkreśleniem, przed każdym z opisów.

Każdy z opisów zostanie użyty tylko raz i każdy z przedstawionych opisów będzie musiał zostać użyty.

- dopóki tablica Portali NIE jest pusta
- wywołanie metody sprawdzRuchGracza()
- powtarzane dla każdego Portalu w tablicy
- pobranie ruchu wykonanego przez gracza
- poproszenie pomocnika gry o wygenerowanie położenia Portalu
- utworzenie trzech obiektów Portal, nadanie im nazw i zapisanie w tablicy ArrayList
- wywołanie naszej metody zakonczGre()
- wykonywanie ustawiającej tego obiektu Portal w celu zapisania jego położenia wygenerowanego przez pomocnik gry
- deklaracja i inicjalizacja niezbędnych zmiennych

```

private void sprawdzRuchGracza(String ruch) {
    iloscRuchow++; 11
    String wynik = "pudło"; 12

    for (Portal portalDoSprawdzenia : listaPortali) { 13
        wynik = portalDoSprawdzenia.sprawdz(ruch); 14
        if (wynik.equals("trafiony")) {
            break; 15
        }
        if (wynik.equals("zatopiony")) {
            listaPortali.remove(portalDoSprawdzenia); 16
            break;
        }
    } // koniec pętli for
    System.out.println(wynik); 17
} // koniec metody

private void zakonczGre() {
    System.out.println("Wszystkie Portale zostały zatopione! Teraz Twoje informacje nie mają znaczenia.");
    if (iloscruchow <= 18) {
        System.out.println("Wykonałeś jedynie " + iloscruchow + " ruchów.");
        System.out.println("Wydostałeś się, zanim Twoje informacje zatonęły.");
    } else {
        System.out.println("Ale się grzebałeś!. Wykonałeś aż " + iloscruchow + " ruchów.");
        System.out.println("Teraz rybki pływają pomiędzy Twoimi informacjami.");
    }
} // koniec metody

public static void main (String[] args) {
    PortalGraMax gra = new PortalGraMax(); 19
    gra.przygotujGre(); 20
    gra.rozpoczniGre(); 21
} // koniec metody
}

```

**Bez względu na to,
co robisz, nie odwracaj
strony!**

**Przynajmniej do
momentu, gdy
skończysz ćwiczenie.**

**Na następnej stronie
podaliśmy naszą wersję
rozwiązania.**

- przerwanie działania pętli, dalsze sprawdzanie jest bezcelowe
- przygotowanie gry
- informacja dla obiektu gry, aby rozpoczął główną pętlę (w której program prosi gracza o podawanie ruchów i sprawdza je)
- początkowo zakładamy, że gracz spudłował (chyba że potem okaże się inaczej)
- nakazanie Portalowi, by sprawdzić ruch i określić, czy zostanie trafiony lub zatopiony
- utworzenie obiektu gry
- wyświetlenie rezultatów ruchu
- wyświetlenie komunikatu dla gracza o uzyskanych wynikach
- ten gość już się utopił, trzeba go zatem usunąć z tablicy Portali i przerwać pętlę
- powtarzane dla każdego Portalu w tablicy
- inkrementacja ilości ruchów wykonanych przez użytkownika

Kod klasy PortalGraMax (gra)

kod przygotowawczy kod testowy kod właściwy

```
import java.util.*;  
public class PortalGraMax {  
  
    private PomocnikGry pomocnik = new PomocnikGry();  
    private ArrayList<Portal> listaPortali = new ArrayList<Portal>();  
    private int iloscRuchow = 0;  
  
    private void przygotujGre() {  
        // najpierw tworzymy portale i określamy ich położenie  
        Portal pierwszy = new Portal();  
        pierwszy.setNazwa("onet.pl");  
        Portal drugi = new Portal();  
        drugi.setNazwa("wp.com");  
        Portal trzeci = new Portal();  
        trzeci.setNazwa("Go2.com");  
        listaPortali.add(pierwszy);  
        listaPortali.add(drugi);  
        listaPortali.add(trzeci);  
  
        System.out.println("Twoim celem jest zatopienie trzech portali.");  
        System.out.println("onet.pl, wp.pl, Go2.com");  
        System.out.println("Postaraj się je zatopić w jak najmniejszej ilości ruchów.");  
  
        for (Portal rozmieszczanyPortal : listaPortali) {  
            ArrayList<String> nowePolozenie = pomocnik.rozmiescPortal(3);  
            rozmieszczanyPortal.setPolaPolozenia(nowePolozenie);  
        } // koniec pętli for  
    } // koniec metody  
  
    private void rozpoczniGre() {  
        while(!listaPortali.isEmpty()) {  
            String ruchGracza = pomocnik.pobierzDaneWejsciowe("Podaj pole:");  
            sprawdzRuchGracza(ruchGracza);  
        } // koniec while  
        zakonczGre();  
    } // koniec metody
```

Deklaracja i inicjalizacja niezbędnych zmiennych.

Utworzenie obiektu ArrayList zawierającego obiekty Portal (innymi słowy na tej liście będzie można zapisywać JEDYNIE obiekty Portal; podobnie jak wyrażenie `Portali` utworzyłoby zwyczajną tablicę obiektów Portal).

Utworzenie trzech obiektów Portal, nadanie im nazw i zapisanie w tablicy ArrayList.

Wyświetlenie krótkich informacji dla gracza.

Powtarzane dla każdego Portalu w tablicy.

Poproszenie pomocnika gry o wygenerowanie położenia Portalu.

Wywołanie metody ustawiającej tego obiektu Portal w celu zapisania jego położenia wygenerowanego przez pomocnika gry.

Dopóki tablica Portali NIE jest pusta (znak ! oznacza przeczenie; odpowiada to wyrażeniu `listaPortali.isEmpty() == false`).

Wywołanie metody sprawdzRuchGracza().

Pobranie ruchu wykonanego przez gracza.

Wywołanie naszej metody zakonczGre().

kod przygotowawczy

kod testowy

kod właściwy

```

private void sprawdzRuchGracza(String ruch) {
    iloscRuchow++; ← Inkrementacja ilości ruchów
    wykonywanych przez użytkownika
    String wynik = "pułapka"; ← Początkowo zakładamy, że gracz spudlował
    (chyba że potem okaże się inaczej)

    for (Portal portalDoSprawdzenia : listaPortali) { ← Powtarzane dla każdego
        Portalu w tablicy
        wynik = portalDoSprawdzenia.sprawdz(ruch); ← Nakazanie Portalowi, by sprawdzić
        ruch i określić, czy został trafiony
        lub zatopiony
        if (wynik.equals("trafiony")) {
            break; ← Przerwanie działania pętli,
            dalsze sprawdzanie jest
            bezcelowe
        }
        if (wynik.equals("zatopiony")) {
            listaPortali.remove(portalDoSprawdzenia); ← Ten gość już się utopił, trzeba go
            zatem usunąć z tablicy Portali
            i przerwać pętlę
            break;
        }
    } // koniec pętli for
    System.out.println(wynik); ← Wyświetlenie rezultatów ruchu
} // koniec metody

private void zakonczGre() {
    System.out.println("Wszystkie Portale zostały zatopione! Teraz Twoje informacje nie mają znaczenia.");
    if (iloscRuchow <= 18) {
        System.out.println("Wykonałeś jedynie " + iloscRuchow + " ruchów.");
        System.out.println("Wydostałeś się, zanim Twoje informacje zatonęły.");
    } else {
        System.out.println("Ale się grzebałeś!. Wykonałeś aż " + iloscRuchow + " ruchów.");
        System.out.println("Teraz rybki płyną pomiędzy Twoimi informacjami.");
    }
} // koniec metody

public static void main (String[] args) {
    PortalGraMax gra = new PortalGraMax(); ← Utworzenie obiektu gry
    gra.przygotujGre(); ← Przygotowanie gry
    gra.rozpoczniGre(); ← Informacja dla obiektu gry, aby
    rozpoczęta główną pętlę (w której
    program prosi gracza o podawanie
    ruchów i sprawdza je)
} // koniec metody
}

```

Wyświetlenie komunikatu dla gracza o uzyskanych wynikach

Ostateczna wersja klasy Portal

```
import java.util.*;  
  
class Portal {  
    private ArrayList<String> polaPolozenia;  
    private String nazwa;  
  
    public void setPolaPolozenia(ArrayList<String> ppol) { ← Składowe obiektu Portal:  
        polaPolozenia = ppol; ← — obiekt ArrayList zawierający pola  
    } ← zajmowane przez ten Portal  
    ← — nazwa Portalu  
  
    public void setNazwa(String nzwPortalu) { ← Metoda służąca do zapisania pokojenia  
        nazwa = nzwPortalu; ← Portalu. (Pokoje są generowane  
    } ← losowo przez metodę rozmiastPortal()  
    ← drugiej wersji klasy PomocnikGry.  
  
    public String sprawdz(String ruch) { ← Prosta metoda ustawiająca  
        String wynik = "pudło"; ← nazwę Portalu.  
        int indeks = polaPolozenia.indexOf(ruch); ← Praktyczne wykorzystanie metody ArrayList.indexOf()  
        if (indeks >= 0) { ← Jeżeli pole wskazane przez gracza jest jednym z pól  
            polaPolozenia.remove(indeks); ← zapisanych w tablicy, to metoda indexOf()wróci jego  
            ← Zastosowanie metody ArrayList.remove()  
            ← do usunięcia elementu tablicy.  
            ← Zastosowanie metody ArrayList.isEmpty()  
            if (polaPolozenia.isEmpty()) { ← w celu sprawdzenia, czy wszystkie pokoje  
                wynik = "zatopiony"; ← zajmowane przez Portal zostały już trafione.  
                System.out.println("Auć! Zatopiłeś portal " + nazwa + " : ( "); ← Informujemy gracza, kiedy  
            } else { ← Portal zostanie zatopiony.  
                wynik = "trafiony"; ←  
            } // koniec if  
        } // koniec if  
        return wynik; ← Zwracamy wynik: "pudło",  
    } // koniec metody ← "trafiony" lub "zatopiony".  
} // koniec klasy
```

Wyrażenia logiczne o bardzo dużych możliwościach

Jak do tej pory wszystkie wyrażenia warunkowe używane w pętlach lub instrukcjach if były bardzo proste. W „Kodzie gotowym do użycia” przedstawionym w dalszej części rozdziału będziemy używać bardziej złożonych wyrażeń logicznych i choć wiemy, że nie zajrzałeś tam, to doszliśmy do wniosku, że to dobry moment na dodanie Twoim wyrażeniom logicznym nieco mocy.

Operatory koniunkcyj i alternatywy (&&, ||)

Załóżmy, że piszesz metodę wybierzAparat() zawierającą wiele reguł określających, jaki aparat fotograficzny należy wybrać. Był może, jesteś w stanie pozwolić sobie na kupno aparatu z przedziału cen od 300 do 4000 PLN, jednak w niektórych przypadkach chcesz ten zakres nieco bardziej zawężić. Na przykład, chciałbyś polecić:

„Jeśli cena mieści się w przedziale pomiędzy 1000 i 1600 PLN, to wybierz aparat X.”

```
if (cena >= 1000 && cena < 1600) {
    aparat = "X";
}
```

Załóżmy, że dysponujesz jakąś szczególną logiką postępowania, która odnosi się jedynie do kilku spośród dziesięciu dostępnych marek aparatów:

```
if (marka.equals("A") || marka.equals("B")) {
    // wykonaj operacje charakterystyczne tylko dla
    // marek A i B
}
```

Wyrażenia logiczne mogą się stać naprawdę rozbudowane i skomplikowane:

```
if ((typZoomu.equals("optyczny") &&
    (stopienPow >= 3 && stopienPow <= 8)) ||
    (typZoomu.equals("cyfrowy") &&
    (stopienPow >= 5 && stopienPow <= 12))) {
    // wykonaj operacje odpowiednie dla danego
    // obiektywu
}
```

Jeśli naprawdę chcesz się zagłębić w szczególne techniczne, to możesz zacząć zwracać uwagę na *pierwszeństwo operatorów*. Jednak nie zachęcamy, abyś stawał się ekspertem w trudnej dziedzinie hierarchii operatorów, zamiast tego polecamy **stosowanie nawiasów**, dzięki którym kod staje się bardziej przejrzysty.

Różne (!= oraz !)

Załóżmy, że w pewnym miejscu nasz program ma działać według następującej zasady: „pewne operacje mają być wykonane dla wszystkich modeli aparatów, za wyjątkiem jednego”.

```
if (model != 2000) {
    // wszystkie aparaty za wyjątkiem modelu 2000
}
```

Lub w przypadku porównywania łańcuchów znaków:

```
if (!marka.equals("X")) {
    // wszystkie marki z wyjątkiem "X"
}
```

Operatory przetwarzania skróconego (&&, ||)

Przedstawione powyżej operatory logiczne — && oraz || — znane są jako operatory **przetwarzania skróconego**. W przypadku operatora && wyrażenie logiczne przyjmie wartość true wyłącznie wtedy, gdy wyrażenia umieszczone po jego obu stronach będą mieć wartość true. A zatem, jeśli wirtualna maszyna Javy stwierdzi, że z lewej strony operatora && pojawiła się wartość false, to już w tym momencie przerywa przetwarzanie całego wyrażenia.

Z kolei w przypadku operatora || wyrażenie będzie mieć wartość true, jeśli jedno z wyrażeń umieszczonych po obu stronach tego operatora przyjmie wartość true. A zatem, jeśli JVM zauważa, że z lewej strony operatora || pojawiła się wartość true, to stwierdzi, że całe wyrażenie ma wartość true i nie będzie zaprzatać sobie głowy sprawdzaniem jego prawej strony.

Dlaczego takie rozwiązanie jest wspaniałe? Założymy, że dysponujesz zmienną referencyjną i nie jesteś pewien, czy została w niej zapisane odwołanie do jakiegoś obiektu. Jeśli spróbujesz wywołać metodę obiektu, posługując się zmienną, w której nie ma żadnego odwołania, to zostanie zgłoszony wyjątek NullPointerException. W takim przypadku spróbuj użyć poniższego kodu:

```
if (zmRef != null && zmRef.czyDobryTyp()) {
    // wykonaj operacje dla zmiennej właściwego typu
}
```

Operatory przetwarzania pełnego (& oraz |)

Operatory & oraz | w przypadku użycia w wyrażenach logicznych działają podobnie jak odpowiadające im operatory && i ||, z tą różnicą, iż wymuszają przetwarzanie całego wyrażenia. Zazwyczaj jednak operatory & i | są używane w innym kontekście — do operacji na bitach.

Gotowy kod: PomocnikGry



Kod gotowy do użycia

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

public class PomocnikGry {
    private static final String alfabet = "abcdefg";
    private int dlugoscPlanszy = 7;
    private int wielkoscPlanszy = 49;
    private int [] plansza = new int[wielkoscPlanszy];
    private int iloscPortali = 0;

    public String pobierzDaneWejsciowe(String komunikat) {
        String daneWejsciowe = null;
        System.out.print(komunikat + " ");
        try {
            BufferedReader is = new BufferedReader(
                new InputStreamReader(System.in));
            daneWejsciowe = is.readLine();
            if (daneWejsciowe.length() == 0) return null;
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
        return daneWejsciowe.toLowerCase();
    }

    public ArrayList rozmiescPortal(int wielkoscPortalu) {
        ArrayList<String> zajetePola = new ArrayList<String>();
        String [] wspolrzedneLnc = new String [wielkoscPortalu];
        String pomoc = null;
        int [] wspolrzedne = new int[wielkoscPortalu];
        int prob = 0;
        boolean powodzenie = false;
        int polozenie = 0;

        iloscPortali++;
        int inkr = 1;
        if ((iloscPortali % 2) == 1) {

            inkr = dlugoscPlanszy;
        }

        while (!powodzenie & prob++ < 200) {
            polozenie = (int) (Math.random() * wielkoscPlanszy);
            //System.out.print(" sprawdź " + polozenie);
            int x = 0;
            powodzenie = true;
            while (powodzenie && x < wielkoscPortalu) {
                if (plansza[polozenie] == 0) {


```

Oto klasa wspomagająca naszą grę. Oprócz metody pobierającej dane wprowadzane przez użytkownika (która wyświetla na ekranie stosowny komunikat i odczytuje dane wprowadzane w wierszu poleceń), klasa ta spełnia jeszcze jedno ważne zadanie — potrafi losowo wybrać pola, jakie ma zajmować Portal. Na Twoim miejscu powolutku wycofalibyśmy się i zostawili ten kod w spokoju, ograniczając się wyłącznie do jego wpisania i skompilowania. Staraliśmy się go skrócić, aby ograniczyć ilość kodu, jaką będziesz musiał wpisać, jednak przez to nie jest on zbytnio czytelny. Pamiętaj, że nie będziesz w stanie skompilować klasy PortalGraMax bez uprzedniego wpisania tej klasy.

Notatka: Aby zdobyć dodatkowe punkty, możesz spróbować usunąć komentarze z wywołań metod `System.out.print()` i `System.out.println()` umieszczone w kodzie metody `rozmiescPortal()` — zobaczysz, jak ona działa. Co prawda usunięcie komentarzy z tych instrukcji pozwoli Ci oszukiwać, gdyż spowoduje wyświetlenie położenia Portali, lecz jednocześnie ututwi Ci przetestowanie metody.

```
        // zawiera współrzędne zapisane jako 'f6'
        // pomocniczy łańcuch znaków
        // bieżące proponowane współrzędne
        // licznik ilości prób
        // flaga = czy znaleziono dobre miejsce
        // bieżące miejsce początkowe

        // n-ty portal do rozmieszczenia
        // określenie przyrostu w poziomie
        // jeśli nieparzysty portal (rozieszczany
        // w pionie)
        // określenie przyrostu w pionie

        // główna pętla poszukiwania
        // wybór losowego punktu początkowego

        // n-ty fragment rozmieszczanego portalu
        // zakładamy, że się udało
        // szukamy sąsiadujących pustych pól planszy
        // jeśli jeszcze nie zajęte
```



Kod gotowy do użycia

Ciąg dalszy klasy PomocnikGry

```

wspolrzedne[x++] = polozenie;                                // zapisujemy miejsce
polozenie += inkrs;                                         // sprawdzamy 'następne' sąsiadujące pole
if (polozenie >= wielkoscPlanszy){                           // poza zakresem - 'dół' planszy
    powodzenie = false;                                       // niepowodzenie
}
if (x>0 & (polozenie % dlugoscPlanszy == 0)) {           // poza zakresem - prawa krawędź planszy
    powodzenie = false;                                       // niepowodzenie
}
} else {                                                       // znalezione pole już jest zajęte
    // System.out.print(" już zajęte " + location);
    powodzenie = false;                                       // niepowodzenie
}
}

}

int x = 0;                                              // zamieniamy na współrzędne
int wiersz = 0;
int kolumna = 0;
// System.out.println("\n");
while (x < wielkoscPortalu) {
    plansza[wspolrzedne[x]] = 1;                            // zaznaczamy pole planszy jako zajęte
    wiersz = (int) (wspolrzedne[x] / dlugoscPlanszy);       // określenie wiersza
    kolumna = wspolrzedne[x] % dlugoscPlanszy;             // pobranie liczby określającej kolumnę
    pomoc = String.valueOf(alfabet.charAt(kolumna));         // konwersja do postaci alfanumerycznej
    zajetePola.add(pomoc.concat(Integer.toString(wiersz)));
    x++;
    // System.out.print(" współrzędne "+x+" = "+zajetePola.get(x-1));
}
// System.out.println("\n");
return zajetePola;
}
}

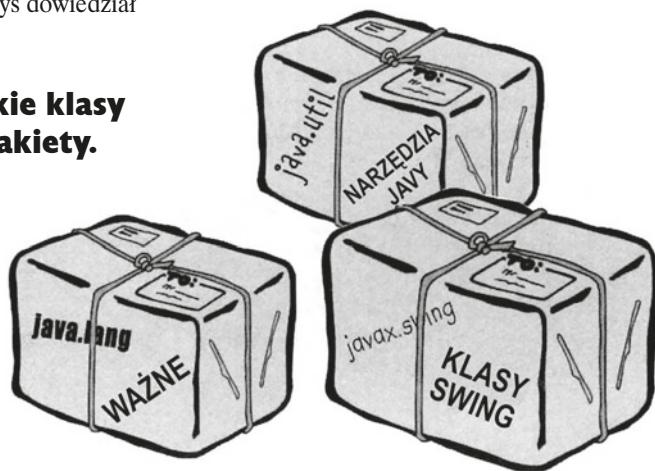
```

Ta instrukcja wyświetla dokładne położenie Portalu.

Stosowanie biblioteki (Java API)

Udało nam się zakończyć pracę nad grą w zatapianie portali. Wszystko dzięki pomocy ze strony klasy `ArrayList`. A teraz, zgodnie z tym, co obiecywaliśmy, nadszedł czas, abyś dowiedział się, jak można „myszkować” po bibliotece Javy.

W Java API wszystkie klasy są zgrupowane w pakiety.



Aby użyć jakieś klasy należącej do API, musisz wiedzieć, w jakim pakucie jest ona umieszczona.

Każda klasa w bibliotece Javy należy do jakiegoś pakietu. Każdy pakiet ma nazwę, na przykład `javax.swing` (ten konkretny pakiet zawiera klasy Swing służące do tworzenia graficznego interfejsu użytkownika, już niedługo dowiesz się o nich więcej). Klasa `ArrayList` została umieszczona w pakiecie `java.util`¹, która, co zapewne będzie dla Ciebie wielką niespodzianką, zawiera grupę klas *użytkowych*. W rozdziale 17. zajdziesz znacznie więcej szczegółowych informacji o pakietach, w tym także o tym, jak można umieszczać własne klasy we własnych pakietach. Jak na razie jednak interesują Cię możliwości wykorzystania klas dostarczanych wraz z Javą.

Wykorzystanie klas należących do Java API we własnym kodzie jest bardzo łatwe. Możesz je potraktować dokładnie w ten sam sposób jak klasy, które tworzysz sam... zupełnie jakbyś je napisał i skompilował, a one czekały i były gotowe do tworzenia obiektów. Istnieje jednak jedna różnica — gdzieś w kodzie programu musisz podać pełną nazwę klasy, jakiej chcesz używać, a to oznacza podanie nazwy pakietu oraz klasy.

W rzeczywistości używasz już klas należących do jednego z pakietów Javy, nawet jeśli sam o tym nie wiedziałeś. System (System.out.println), String oraz Math — wszystkie te klasy należą do pakietu java.lang.

¹ „util” to początkowa część angielskiego słowa „utility” oznaczającego, między innymi, użyteczną rzecz.

Nie istnieja
głupie pytania

Musisz znać pełną nazwę klasy*, której chcesz użyć w swoim kodzie.

ArrayList nie jest pełną nazwą klasy ArrayList, tak jak „Kasia” stanowi jedynie imię, a nie pełne personalia osoby (no chyba że „Kasia” to ktoś w stylu Madonny lub Cher, ale nie będziemy posuwać się aż tak daleko). Pełna nazwa klasy ArrayList ma następującą postać:



Musisz poinstruować Javę, której klasy ArrayList chcesz używać. Masz do wyboru dwie możliwości:

A Import

Na początku pliku z kodem źródłowym umieść instrukcję import:

```
import java.util.ArrayList;
public class MojaKlasa { ... }
```

lub

B Wpisanie

Możesz także podawać pełną nazwę klasy we wszystkich miejscach kodu. Za każdym razem, gdy będziesz jej używać. Gdziekolwiek byś tego nie zrobił.

W przypadku deklarowania oraz tworzenia obiektu klasy:

```
java.util.ArrayList<Pies> lista = new java.util.ArrayList<Pies>();
```

W przypadku używania klasy do określenia typu argumentu:

```
public void jazda(java.util.ArrayList<Pies> lista) {...}
```

W przypadku używania klasy jako typu wyniku zwracanego przez metodę:

```
public java.util.ArrayList<Pies> metodaPomocnicza() {...}
```

* chyba że klasa ta należy do pakietu java.lang

P: Dlaczego konieczne jest podawanie pełnej nazwy? Czy jest to jedyny powód tworzenia pakietów?

O: Pakiety są istotne z trzech podstawowych powodów. Po pierwsze, pomagają w ogólnej organizacji projektu lub biblioteki. Zamiast zarządzania setką klas, można je pogrupować w pakiety zawierające klasy o określonym przeznaczeniu lub możliwościach funkcjonalnych (na przykład związane z graficznym interfejsem użytkownika, strukturami danych, obsługą baz danych i tak dalej).

Po drugie, pakiety określają przestrzenie nazw, dzięki którym można uniknąć konfliktów, gdy nagle Ty oraz 12 innych programistów w Twojej firmie zdecydujecie się stworzyć klasę o tej samej nazwie. Jeśli stworzysz klasę o nazwie Set, a klasa o tej samej nazwie zostanie także stworzona przez kogoś innego (lub będzie należeć do Java API), to jednocześnie będzie Ci potrzebna jakaś metoda poinformowania JVM o tym, której z tych klas chcesz użyć.

Po trzecie, pakiety zapewniają pewien poziom bezpieczeństwa, gdyż można wprowadzić ograniczenie, by jedynie klasy należące do tego samego pakietu miały dostęp do tworzonego kodu. Z tymi zagadnieniami zapoznasz się w rozdziale 17.

P: No dobrze, wróćmy do zagadnienia kolizji nazw. W jaki sposób może nam pomóc pełna nazwa klasy? W jaki sposób można zagwarantować, że dwie osoby nie stworzą pakietów o tej samej nazwie?

O: W Javie stosowana jest konwencja nazewnictwa, która zazwyczaj zapobiega takim sytuacjom. Oczywiście jeśli programiści stosują się do jej zaleceń. Zagadnienia te zostały szczegółowo opisane w rozdziale 17.

Skąd pochodzi litera „x”?

(lub co oznacza nazwa pakietu zaczynająca się od javax?)

W pierwszej i drugiej wersji języka Java (1.02 oraz 1.1) wszystkie dostarczane klasy (czyli, innymi słowy, standardowa biblioteka języka) znajdowały się w pakietach, których nazwy rozpoczynały się od **java**. Oczywiście zawsze istniał pakiet **java.lang**, którego nie trzeba importować. Istniały pakiety **java.net**, **java.io**, **java.util** (choć w tamtych czasach nie istniała klasa **ArrayList**) oraz kilka innych, w tym **java.awt** — pakiet gromadzący klasy używane do tworzenia graficznego interfejsu użytkownika.

Na horyzoncie pojawiały się jednak także i inne pakiety, które nie były dołączane do standardowej biblioteki Javy. Były one nazywane **rozszerzeniami** (ang. *extensions*) i dzieliły się na dwa rodzaje: standardowe i niestandardowe. Standardowymi były te rozszerzenia, które firma Sun uznała za oficjalne. Pozostałe pakiety — eksperymentalne, testowe oraz wszelkie pozostałe — mogły w ogóle nie ujrzeć światła dzennego.

Na mocy przyjętej konwencji w nazwach pakietów uznanych za rozszerzenia standardowe, do słowa „java” dodawana była litera „x”. Prekursorem wszystkich rozszerzeń standardowych była biblioteka Swing. Zawierała ona kilka pakietów, których nazwy zawsze rozpoczynały się od **javax.swing**.

Jednak rozszerzenia standardowe mogą zostać dołączone do standardowej biblioteki Javy rozpowszechnianej wraz z językiem. Właśnie tak stało się z biblioteką Swing, która została dołączona do standardowej biblioteki Javy, zaczynając od wersji 1.2 języka (która stała się pierwszą wersją określana jako „Java 2”). Wszyscy (włącznie z Tobą) pomyśleli sobie: „Super! Teraz każdy, kto ma Javę, będzie także mieć bibliotekę Swing i nie będziemy musieli się przejmować tym, jak zainstalować te klasy u naszych klientów”.

Jednak gdzieś tam czał się problem. Otóż kiedy pakiet był promowany i stawał się „oficjalnym”, trzeba było zmienić początek jego pełnej nazwy z **javax** na **java**. Przecież to OCZYWISTE. Każdy wie, że pakiety należące do standardowej biblioteki Javy nie mają litery „x” w nazwie oraz że tylko nazwy rozszerzeń zaczynają się od **javax**. A zatem tuż przed udostępnieniem ostatecznej wersji Javy 1.2 (i mówiąc „tuż”, mamy na myśli „bezpośrednio”) Sun zmienił nazwy pakietów i usunął z nich literę „x” (oraz wprowadził inne zmiany). Napisano książki i artykuły, w których pojawił się kod wykorzystujący bibliotekę Swing i nowe nazwy pakietów. Konwencje nazewnictwa zostały zachowane. Wszystko było w porządku.

I wtedy około 20 tysięcy zrozpaczonych programistów uświadomiło sobie, że ta prosta zmiana nazwy pakietów przyczyniła się do prawdziwej katastrofy! Cały kod wykorzystujący klasy biblioteki Swing trzeba było zmodyfikować! Dramat! Wyobraź sobie wszystkie te instrukcje import zawierające **javax**...

I w ostatniej chwili zdesperowanym programistom, których nadzieję zaczynały się już rozwiewać, udało się przekonać firmę Sun do „zmiany konwencji i zachowania istniejącego kodu”. A zatem, kiedy w standardowej bibliotece Javy zobaczyłeś pakiety, których nazwy zaczynają się od **javax**, będziesz już wiedział, że początkowo pojawiły się one jako rozszerzenia, a potem zostały dołączone do Java API.



CELNE SPOSTRZEŻENIA

- **ArrayList** to klasa należąca do Java API.
- Aby umieścić coś w obiekcie **ArrayList**, należy wywołać metodę **add()**.
- Aby usunąć coś z obiektu **ArrayList**, należy wywołać metodę **remove()**.
- Aby określić położenie obiektu w **ArrayList** (lub sprawdzić, czy w ogóle jest dostępny), należy wywołać metodę **indexOf()**.
- Aby sprawdzić, czy obiekt **ArrayList** jest pusty, należy wywołać metodę **isEmpty()**.
- Aby określić wielkość obiektu **ArrayList** (czyli ilość przechowywanych w nim elementów), należy wywołać metodę **size()**.
- Pamiętaj, że w przypadku normalnych tablic, ich **długość** (ilość elementów, jakie tablica zawiera) można określić, posługując się **składową length**.
- Obiekt **ArrayList** **dynamicznie zmienia swoją wielkość**, zgodnie z bieżącymi potrzebami. Powiększa się w przypadku dodawania obiektów i **zmniejsza**, kiedy są one usuwane.
- Deklarując obiekt **ArrayList**, należy użyć **parametru typu**, czyli nazwy typu umieszczonej w nawiasach kątowych. Na przykład: **ArrayList<Button>** oznacza, że w obiekcie **ArrayList** będzie można zapisywać wyłącznie obiekty klasy **Button** (lub klas potomnych, o czym przekonasz się w kolejnych rozdziałach).
- Choć obiekty **ArrayList** przechowują obiekty, a nie wartości typów podstawowych, to kompilator, dodając do listy wartość typu podstawowego, automatycznie umieszcza ją w obiekcie (a przy pobieraniu — odczytuje jej wartość z tego obiektu). (Więcej informacji na ten temat znajdziesz w dalszej części książki).
- Klasa są grupowane w pakiety.
- Klasa mają pełną nazwę stanowiącą połączenie nazwy pakietu oraz nazwy klasy. **ArrayList** to w rzeczywistości klasa **java.util.ArrayList**.
- Aby zastosować klasę znajdującej się w dowolnym pakiecie oprócz **java.lang**, należy podać jej pełną nazwę.
- Można użyć bądź to instrukcji **import** umieszczonej na początku kodu źródłowego, bądź też podawać pełną nazwę klasy w kodzie.

Nie istnieja głupie pytania

P: Czy instrukcja import sprawia, że klasa staje się większa? Czy sprawia ona, że podczas komplikacji wskazana klasa lub pakiet zostaje dołączona do mojej klasy?

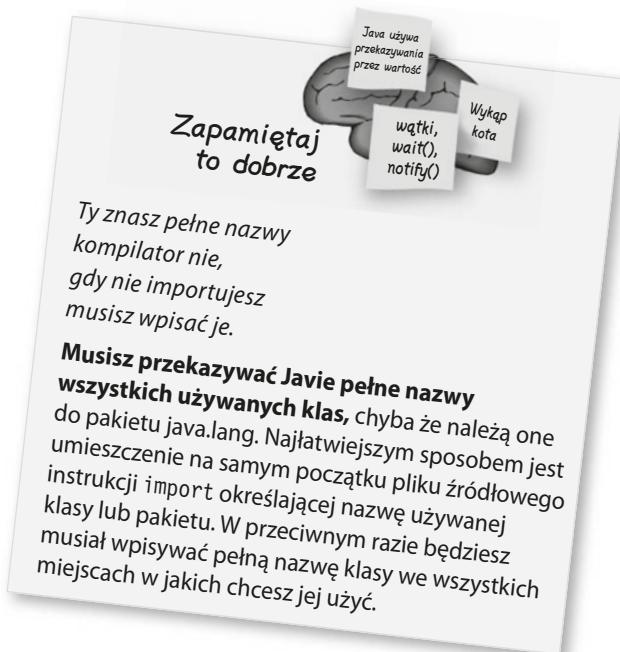
O: Zapewne wcześniej używałeś języka C? Instrukcja import to nie to samo co include. A zatem odpowiedź na oba powyższe pytania brzmi — Nie. Powtórz za mną: „Instrukcja import zaoszczędza nam jedynie trochę pisania”. I nic więcej, naprawdę. Nie musisz obawiać się, że w wyniku użycia wielu instrukcji import Twój kod stanie się większy lub wolniejszy. Instrukcja ta stanowi jedynie sposób przekazania kompilatorowi Javy pełnej nazwy klasy.

P: W porządku. To dlaczego nigdy nie muszę stosować tej instrukcji w przypadku korzystania z klasy String lub System?

O: Pamiętaj, że cała zawartość pakietu java.lang jest, można by rzec, „zawczasu importowana”. Ponieważ klasy należące do tego pakietu są tak istotne, nie trzeba podawać ich pełnych nazw. Istnieje tylko jedna klasa String i jedna klasa System, a Java doskonale wie, gdzie należy ich szukać.

P: Czy swoje własne klasy też muszę umieszczać w pakietach?

O: W realnym świecie (którego powinieneś unikać) raczej będziesz chciał umieszczać własne klasy w pakietach. Zagadnienie to opisaliśmy szczegółowo w rozdziale 17. Jak na razie jednak nie będziemy umieszczać prezentowanych przykładów w pakietach.



Na wszelki wypadek powtórzmy jeszcze raz, gdybyś jakimś dziwnym zbiegiem okoliczności jeszcze tego nie pamiętał:



„Dobrze jest wiedzieć, że w pakiecie `java.util` istnieje klasa `ArrayList`. Ale jak mogłabym się o tym sama dowiedzieć?”

— Julia, lat 31, modelka

Jak poznać API?

Interesują nas dwa zagadnienia:

- 1 Jakie klasy należą do biblioteki Javy?
- 2 Kiedy już znajdziemy klasę — to jak poznać jej możliwości?



Skorzystaj z dokumentacji API dostarczanej w formie plików HTML

The screenshot shows a Mozilla Firefox browser window displaying the Java 2 Platform Standard Edition 5.0 API documentation. The URL in the address bar is <http://java.sun.com/j2se/1.5.0/docs/api/>. The page title is "Overview (Java 2 Platform SE 5.0) - Mozilla Firefox". The main content area is titled "ORACLE APIs and Documentation on SDN" and includes a navigation bar with links for APIs, Downloads, Products, Support, Training, and Participate. Below the navigation bar, there are links for Overview, Package, Class, Use, Tree, Deprecated, Index, Help, PREV, NEXT, FRAMES, and NO FRAMES. A "Java™ 2 Platform Standard Edition 5.0 API Specification" section is present, along with a "Description" section. The main content area is titled "Java 2 Platform Packages" and lists several packages with their descriptions:

Package	Description
<code>java.applet</code>	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
<code>java.awt</code>	Contains all of the classes for creating user interfaces and for painting graphics and images.
<code>java.awt.color</code>	Provides classes for color spaces.
<code>java.awt.datatransfer</code>	Provides interfaces and classes for transferring data between and within applications.
<code>java.awt.dnd</code>	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
<code>java.awt.event</code>	Provides interfaces and classes for dealing with different types of events fired by AWT components.
<code>java.awt.font</code>	Provides classes and interface relating to fonts.
<code>java.awt.geom</code>	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
<code>java.awt.image</code>	Provides classes and interfaces for the input method framework.
<code>java.awt.im</code>	Provides interfaces that enable the development of input methods that can be used with any Java runtime.

Skorzystaj z dokumentacji API dostarczanej w formie dokumentów HTML

Java jest dostarczana wraz ze wspaniałą dokumentacją, która, co jest nieco zaskakujące, jest nazywana Java API. Stanowi ona część większego zbioru informacji określonego jako dokumentacja Java 5 Standard Edition (choć zależnie od dnia tygodnia, w jakim zajrzysz na witrynę firmy Oracle, może ona także nosić nazwę „Java 2 Standard Edition 5.0”). Dokumentację tę trzeba pobrać osobno, gdyż nie jest ona dołączana do plików instalacyjnych Javy. Jeśli masz szybkie łącze z internetem lub bardzo dużo cierpliwości, to możesz przeglądać tę dokumentację bezpośrednio na witrynie <http://download.oracle.com/javase/index.html>. Ale uwierz nam — naprawdę będziesz chciał mieć ją u siebie na dysku.

Dokumentacja API jest najlepszym źródłem pozwalającym na zdobycie szczegółowych informacji o klasach i ich metodach. Założmy, że przeglądałeś książkę o Javie i znalazłeś w niej klasę o nazwie `Calendar` należącą do pakietu `java.util`. W książce nie było jednak zbyt wielu informacji na jej temat — ot tyle, abyś doszedł do wniosku, że jest to klasa, która będzie Ci potrzebna, za mało jednak, abyś dobrze poznął jej metody.

Na przykład książka zawiera informacje na temat argumentów metod oraz o zwracanych przez nie wartościach. Sprawdź klasę `ArrayList`. W książce znajdziesz dane o metodzie `indexOf()`, której używaliśmy przy tworzeniu klasy `Portal`. Niemniej jednak, nawet jeśli wiesz o istnieniu metody `indexOf()` oraz o tym, że wymaga ona przekazania obiektu i zwraca jego indeks (liczbę typu `int`), wciąż będziesz musiał dowiedzieć się czegoś o kluczowym zagadnienu — co się stanie, jeśli przekazanego obiektu nie będzie w tablicy `ArrayList`. Analiza sygnatury metody niewiele Ci w tym przypadku pomoże. Znacznie bardziej pomocna okaże się dokumentacja API (przynajmniej w większości przypadków). Dokumentacja API informuje, że w przypadku gdy obiekt przekazany do metody `indexOf()` nie zostanie znaleziony, metoda ta zwróci wartość `-1`. To dzięki tym informacjom wiemy, że możemy użyć tej metody zarówno w celu sprawdzenia, czy obiekt w ogóle jest zapisany w tablicy `ArrayList`, jak również do określenia jego indeksu (jeśli obiekt jest dostępny). Bez tych informacji, moglibyśmy wyobrażać sobie, że w przypadku, gdy poszukiwany obiekt nie jest dostępny, wywołanie metody `indexOf()` doprowadzi obiekt `ArrayList` do „wybuchu”.

1

Przejrzyj listę pakietów i wybierz (kliknij) jeden z nich, aby w dolnej ramce wyświetlić wyłącznie listę klas z danego pakietu.

2

Przejrzyj listę klas i wybierz (kliknij) jedną z nich, aby w głównej ramce wyświetlić informacje na jej temat.

To właściwie tu znajdują się wszystkie przydatne informacje. Możesz przejrzeć listę zawierającą skrócone informacje o metodach lub kliknąć jedną z metod, aby wyświetlić pełne informacje na jej temat.



Ćwiczenie



Magnesiki z kodem

Czy możesz poustawić magnesiki z kodem w taki sposób, by utworzyć działający program, generujący przedstawione poniżej wyniki? Uwaga: Aby wykonać zadanie, potrzebujesz PEWNEJ nowej informacji — jeśli zajrzesz do dokumentacji klasy ArrayList, znajdziesz w niej drugą wersję metody add, która wymaga podania dwóch argumentów:

```
add(int index, Object o)
```

Pozwala ona określić miejsce listy, w jakim należy umieścić dodawany obiekt.

```
a.remove(2);  
printAL(a);
```

```
printAL(a);
```

```
if (a.contains("dwa")) {  
    a.add("2.2");  
}
```

```
public static void wyswietlWszystko(ArrayList<String> a) {
```

```
    a.add(2,"dwa");
```

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

```
        System.out.print(element + " ");
```

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

```
    a.add(0,"zero");  
    a.add(1,"jeden");
```

```
    if (a.contains("trzy")) {  
        a.add("cztery");  
    }
```

```
    public class ArrayListMagnet {
```

```
        if (a.indexOf("pięć") != 4) {  
            a.add(4, "4.2");  
        }
```

```
}
```

```
}
```

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

```
wyswietlWszystko(a);
```

Wiersz poleceń

```
T:\>java ArrayListMagnet  
zero jeden dwa trzy  
zero jeden trzy cztery  
zero jeden trzy cztery 4.2  
zero jeden trzy cztery 4.2
```

```
ArrayList<String> a = new ArrayList<String>();
```

```
for (String element : a) {
```

```
    a.add(3,"trzy");  
    wyswietlWszystko(a);
```





Ćwiczenie rozwiążanie

Wiersz polecenia

```
T:\>java ArrayListMagnet
zero jeden dwa trzy
zero jeden trzy cztery
zero jeden trzy cztery 4.2
zero jeden trzy cztery 4.2
```

```
import java.util.*;
public class ArrayListMagnet {
```

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

```
        ArrayList<String> a = new ArrayList<String>();
```

```
        a.add(0,"zero");
        a.add(1,"jeden");
```

```
        a.add(2,"dwa");
```

```
        a.add(3,"trzy");
        wyswietlWszystko(a);
```

```
        if (a.contains("trzy")) {
            a.add("cztery");
        }
```

```
        a.remove(2);
```

```
        wyswietlWszystko(a);
```

```
        if (a.indexOf("cztery") != 4) {
            a.add(4, "4.2");
        }
```

```
        printAL(a);
```

```
        if (a.contains("dwa")) {
            a.add("2.2");
        }
```

```
        printAL(a);
    }
```

```
    public static void printAL(ArrayList<String> al) {
```

```
        for (String element : al) {
```

```
            System.out.print(element + " ");
        }
```

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

```
}
```

7. Dziedziczenie i polimorfizm

Wygodniejsze życie w Obiekcie



Do momentu, kiedy spróbowaliśmy Planu Polimorfizmu, byliśmy źle opłacanymi, przepracowanymi koderami. Ale dzięki Planowi nasza przyszłość rysuje się w jasnych barwach. Także Ty możesz mieć takie perspektywy.

Planuj swoje programy, myśląc perspektywicznie. Gdyby istniała metoda pisania programów w Javie w taki sposób, abyś mógł brać więcej urlopu, jaką miałaby dla Ciebie wartość? Co by było, gdybyś mógł tworzyć swój kod w taki sposób, że ktoś *inny* mógłby go **łatwo** rozszerzać? I gdybyś mógł pisać kod bardzo elastyczny, pozwalający na wprowadzanie tych denerwujących zmian specyfikacji zgłaszanych w ostatniej chwili, czy byłbyś zainteresowany takimi możliwościami? Oto nadszedł Twój szczęśliwy dzień. Możesz to mieć — za trzy niewielkie wpłaty o równowartości 60 minut pracy. Gdy zdobędziesz Plan Polimorfizmu, poznasz pięć kroków do lepszego projektowania klas, trzy sztuczki polimorficzne i osiem sposobów tworzenia elastycznego kodu, a jeśli zadzwoniš juž teraz — otrzymasz dodatkową lekcję na temat czterech sposobów stosowania dziedziczenia. Nie wahaj się, tak doskonała oferta zapewni Ci swobodę projektowania i elastyczność, na jaką zasługujesz. Jest szybka, łatwa i dostępna od ręki. Zacznię juž dziś, a dorzucimy Ci dodatkowy poziom abstrakcji.

Wojna o fotel raz jeszcze...

Pamiętasz może zamieszczoną „wieki temu” — w rozdziale 2. — opowieść o Bronku „proceduralnym” i Jurku „obiektowcu”, którzy rywalizowali o Superfotel™? Przejrzyjmy jeszcze raz fragmenty tej opowieści, aby przypomnieć sobie podstawy dziedziczenia.

BRONEK: W twoim programie powtarzają się te same fragmenty kodu! Procedura do obracania jest we wszystkich czterech Figurach. Idiotyczny projekt. Musisz mieć aż cztery różne „metody” do obracania. Czy taki projekt może być dobry?

JUREK: O! Jak mniemam, nie widziałeś ostatniej wersji projektu. Pozwól, że pokaże ci, jak w programowaniu obiektowym działa **dziedziczenie**.

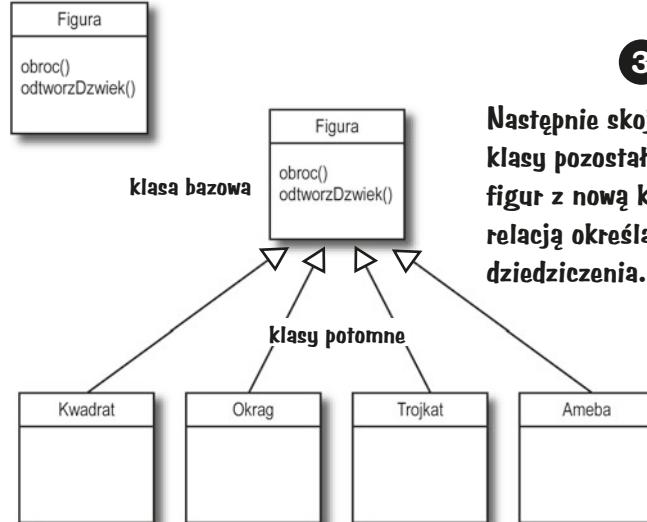


1

Zauważłem wspólnie elementy wszystkich czterech klas.

2

Wszystko to są Figury i mogą się obracać i odtwarzać dźwięki. A zatem wyróżniłem i wydzieliłem ich wspólne cechy i umieściłem je w nowej klasie o nazwie Figura.



3

Następnie skojarzyłem klasy pozostałych czterech figur z nową klasą Figura relacją określana mianem dziedziczenia.

Możesz to rozumieć jako: „Kwadrat dziedziczy po klasie Figura”, „Okrag dziedziczy po klasie Figura” i tak dalej. Metody obroc() i odtworzDzwiek() usunąłem z klas konkretnych figur, dzięki czemu aktualnie nie są już powielane.

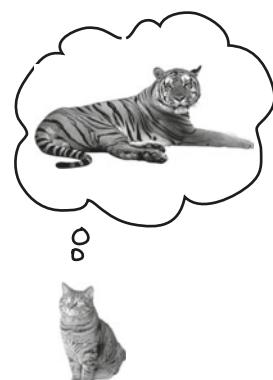
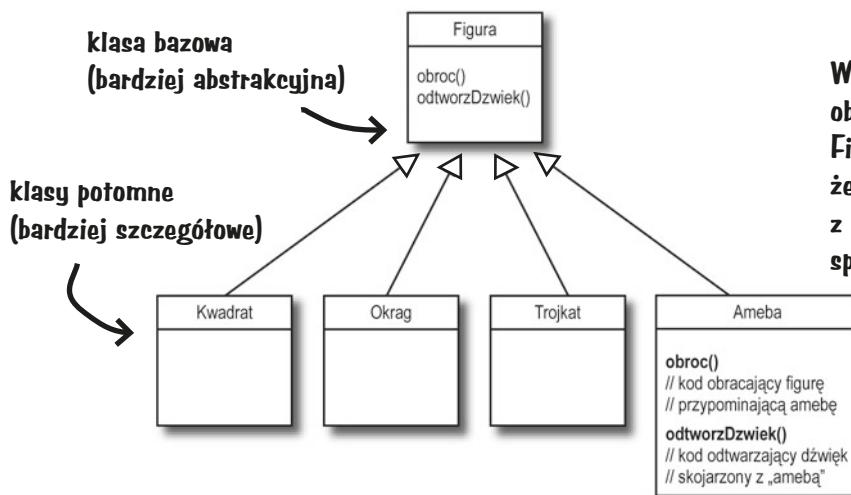
Klasa Figura nazywana jest klasą **bazową** pozostały czterech klas. Z kolei pozostałe cztery klasy są klasami **potomnymi** klasy Figura. Klasa potomne dziedziczą metody klasy bazowej. Innymi słowy, jeśli klasa Figura dysponuje pewnymi możliwościami funkcjonalnymi, to jej klasy potomne automatycznie uzyskują te same możliwości funkcjonalne.

A co z metodą obroc() klasy Ameba?

BRONEK: Ale czy cały problem nie polegał właśnie na tym, że klasa Ameba musiała mieć zupełnie inne metody obroc() i odtworzDzwiek()?

W jaki sposób ameba może zrobić coś innego, skoro dziedziczy swoje możliwości funkcjonalne po klasie Figura?

JUREK: To ostatni krok. Klasa Ameba *przesłania* metody klasy Figura. Później, w trakcie działania programu, wirtualna maszyna Javy dokładnie „wie”, którą metodę obroc() należy wywołać, kiedy ktoś będzie chciał obrócić obiekt Ameba.



WYSIL SZARE KOMÓRKI

W jaki sposób, wykorzystując strukturę dziedziczenia, przedstawiłbyś domowego kotka i tygrysa? Czy domowy kotek to wyspecjalizowana wersja tygrysa? Które zwierzę byłoby klasą bazową, a które potomną? A może oba zwierzaki są klasami potomnymi jakiejś *innej* klasy?

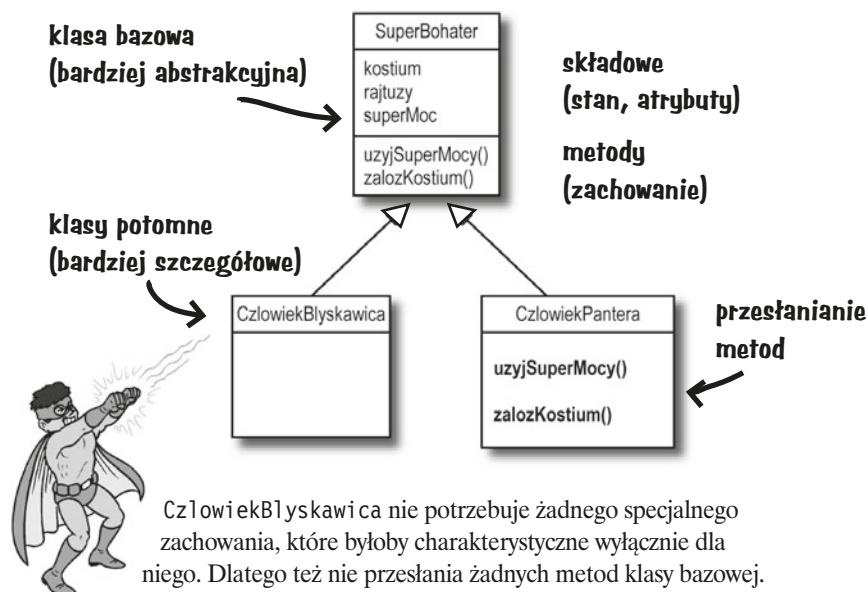
W jaki sposób zaprojektowałbyś strukturę dziedziczenia? Jakie metody zostałyby przesłonięte?

Pomyśl o tym. *Zanim* przejdziesz na następną stronę.

Zrozumienie dziedziczenia

Projektując klasy przy wykorzystaniu dziedziczenia, powinieneś umieścić wspólny kod w klasie bazowej, a następnie poinformować klasy wyspecjalizowane, iż konkretna wspólna (bardziej abstrakcyjna) klasa, jest ich klasą bazową. W przypadku, gdy jedna klasa dziedziczy po drugiej, **to klasa potomna dziedziczy po klasie bazowej**.

W Javie mówimy, że **klasa potomna rozszerza klasę bazową**. Relacja dziedziczenia oznacza, że klasa potomna dziedziczy **składowe i metody** klasy bazowej. Na przykład, jeśli CzlowiekPantera jest klasą potomną klasy SuperBohater, to automatycznie odziedziczy ona wszystkie składowe i metody tej klasy, w tym kostium, rajtuzy, superMoc, uzyjSuperMocy() i tak dalej. Jednak w **klasie potomnej** CzlowiekPantera **można teraz dodać nowe**, unikalne dla niej, **metody i składowe**; można także **przesłonić metody odziedziczone po klasie bazowej** SuperBohater.



CzlowiekBlyskawica nie potrzebuje żadnego specjalnego zachowania, które byłoby charakterystyczne wyłącznie dla niego. Dlatego też nie przesłania żadnych metod klasy bazowej.

Metody i składowe klasy SuperBohater są w tym przypadku całkowicie wystarczające. Z kolei CzlowiekPantera ma konkretne wymagania dotyczące jego kostiumu oraz mocy; z tego względu metody uzyjSuperMocy() oraz zalozKostium() zostały w tej klasie przesłonięte.

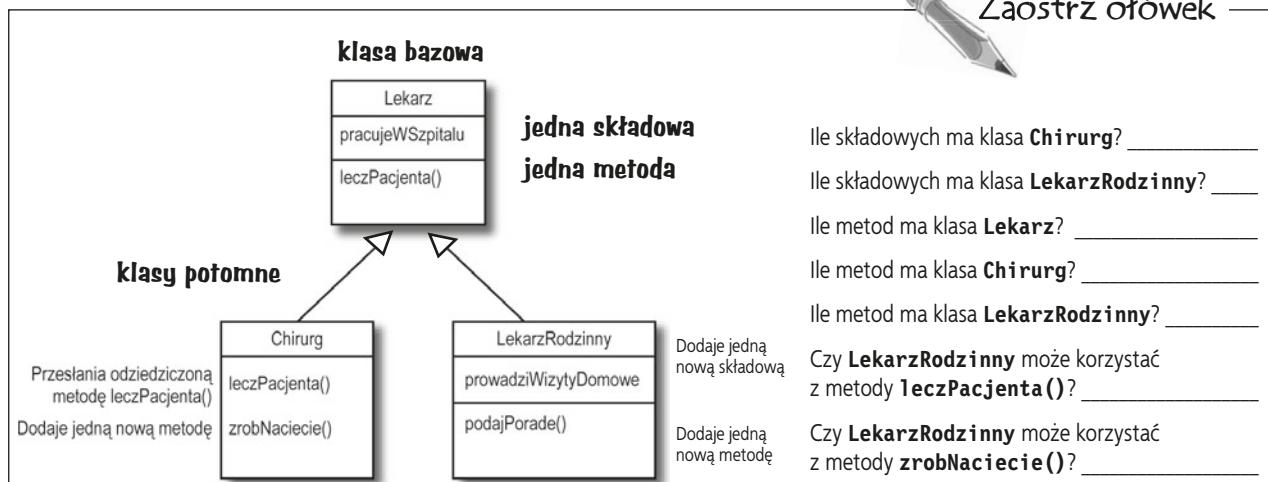
Składowe nie są przesłaniane, gdyż nie ma takiej potrzeby. Nie określają one żadnego specjalnego działania, zatem klasa potomna może im nadawać dowolne wartości. Klasa CzlowiekPantera może określić, że odziedziczone rajtuzy są purpurowe, a klasa CzlowiekBlyskawica, że są one białe.

Przykład dziedziczenia

```
public class Lekarz {
    boolean pracujeWSzpitalu;
    void leczPacjenta() {
        // przeprowadza badanie
    }
}
```

```
public class LekarzRodzinny extends Lekarz {
    boolean prowadziWizytyDomowe;
    void podajPorade() {
        // wybiera i podaje poradę
    }
}
```

```
public class Chirurg extends Lekarz {
    void leczPacjenta() {
        // przeprowadza rozpoznanie
    }
    void zrobNaciecie() {
        // robi nacięcie (auuu!!)
    }
}
```



Ille składowych ma klasa **Chirurg**? _____

Ille składowych ma klasa **LekarzRodzinny**? _____

Ille metod ma klasa **Lekarz**? _____

Ille metod ma klasa **Chirurg**? _____

Ille metod ma klasa **LekarzRodzinny**? _____

Czy **LekarzRodzinny** może korzystać z metody **leczPacjenta()**? _____

Czy **LekarzRodzinny** może korzystać z metody **zrobNaciecie()**? _____

Zaprojektujmy drzewo dziedziczenia dla programu symulacji zwierząt

Wyobraź sobie, że zostałeś poproszony o zaprojektowanie programu symulacyjnego, który pozwala użytkownikowi na umieszczenie w środowisku testowym kilku różnych zwierząt i sprawdzenie, co się stanie. Nie musimy już teraz tworzyć kodu programu, na razie bardziej interesuje nas jego projekt.

Otrzymaliśmy listę *niektórych* — lecz nie wszystkich — zwierząt, które będą dostępne w programie. Wiemy, że każde ze zwierząt będzie reprezentowane przez obiekt oraz że obiekty zwierząt będą się poruszały po obszarze środowiska, robiąc to, do czego zostały zaprogramowane.

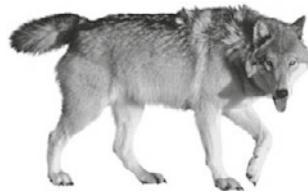
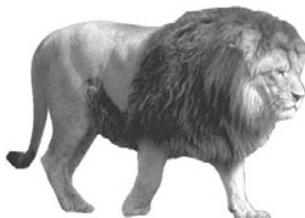
Poza tym chcemy, aby w każdej chwili można było dodawać do programu nowe typy zwierząt.

W pierwszej kolejności chcemy określić wspólne, abstrakcyjne cechy, które posiadają wszystkie zwierzęta i umieścić je w pewnej klasie bazowej.

- 1 Poszukaj obiektów mających wspólnie atrybuty i zachowania.

Jakie wspólne cechy ma sześć przedstawionych typów obiektów? To pomoże Ci wyróżnić wspólne zachowania (krok 2.).

W jaki sposób wszystkie te typy są powiązane ze sobą? To pomoże Ci określić drzewo relacji dziedziczenia (kroki 4. i 5.).



Zastosowanie dziedziczenia w celu uniknięcia powielania kodu w klasach potomnych

Dysponujemy pięcioma *składowymi*:

zdjecie — nazwa pliku zawierającego zdjęcie zwierzęcia zapisane w formacie JPEG.

pożywienie — typ pożywienia preferowanego przez dane zwierzę.
Jak na razie może przyjmować tylko dwie wartości: *mięso* lub *trawa*.

glod — liczba całkowita reprezentująca poziom głodu danego zwierzęcia.
Jej wartość zmienia się w zależności od tego, kiedy (i jak dużo) zwierzę je.

terytorium — wartości reprezentujące wysokość i szerokość „obszaru”
(na przykład 640×480), w jakim zwierzę będzie się poruszać.

polozenie — współrzędne x i y miejsca, w którym zwierzę się znajduje.

Dodatkowo mamy cztery **metody**:

halasuj() — czynność wykonywana, gdy zwierzę ma hałasować.

jedz() — czynność wykonywana, gdy zwierzę napotka swoje ulubione źródło pożywienia — *mięso* lub *trawa*.

spij() — czynność wykonywana, gdy zwierzę ma spać.

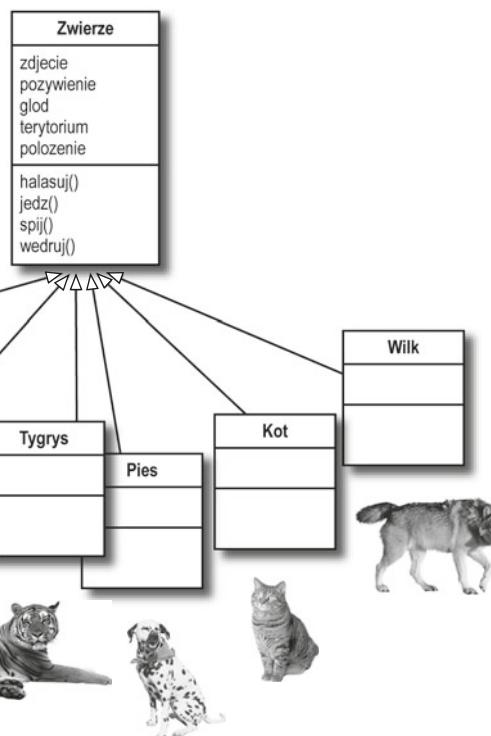
wedruij() — czynność wykonywana, gdy zwierzę ani nie je, ani nie śpi
(czyli, prawdopodobnie, przemieszcza się po swoim terytorium, oczekując
na spotkanie ze źródłem pożywienia lub granicą zajmowanego obszaru).

2

Zaprojektuj klasę reprezentującą wspólny stan oraz zachowanie zwierząt.

Wszystkie obiekty są zwierzętami, a zatem wspólnej klasie bazowej nadamy nazwę **Zwierze**.

Umieścimy w niej wszystkie składowe i metody, których zwierzę może potrzebować.



Czy wszystkie zwierzęta pożywiają się w ten sam sposób?

Załóżmy, że zgadzamy się co do jednego — przedstawione składowe spełniają wymagania wszystkich typów zwierząt. Lew będzie mieć własne wartości składowych zdjecie, pozywienie (w tym przypadku sądzimy, że będzie to *mięso*), głod, terytorium i położenie.

Hipopotam będzie mieć inne wartości składowych, jednak same składowe będą takie same jak we wszystkich pozostałych typach zwierząt. To samo dotyczy psa, tygrysa i tak dalej. Jednak co z zachowaniem poszczególnych zwierząt?

Jakie metody należy przesłonić?

Czy lew **hałasuje** w taki sam sposób jak pies? Czy kot **pożywia** się tak samo jak hipopotam? Być może w Twojej wersji programu tak jest, jednak w naszej jedzenie i hałasowanie to czynności charakterystyczne dla każdego z typów zwierząt. Nie jesteśmy w stanie określić, jak można by napisać te metody w taki sposób, aby mogły być używane dla każdego rodzaju zwierzęcia. No dobrze, to nieprawda. Moglibyśmy napisać metodę **halasuj()** w taki sposób, aby odtwarzała plik dźwiękowy określony za pomocą składowej obiektu i unikalny dla poszczególnych typów zwierząt; jednak w ten sposób metoda nie byłaby specjalizowana. Niektóre zwierzęta mogą wydawać różne dźwięki w różnych sytuacjach (na przykład w przypadku odnalezienia pożywienia lub spotkania z nieprzyjacielem i tak dalej).

A zatem podobnie jak było w przypadku klasy Ameba, która przesłaniała metodę **obroc()** bazowej klasy Figura, aby zapewnić działanie charakterystyczne dla ameby (innymi słowy — *unikalne*), będziemy musieli postąpić podobnie w klasach poszczególnych zwierząt.

Zwierze
zdjecie
pozywienie
głod
tertorium
położenie
halasuj()
jedz()
spij()
wedruj()

Jestem okrutnym roślinożercą.



W społeczności psów szczenięcie jest bardzo ważnym czynnikiem naszej kulturalnej tożsamości. Postugujemy się unikalnymi dźwiękami i chcemy, aby ta różnorodność dźwięków była dostępna i traktowana z szacunkiem.



Lepiej przestonić te dwie metody — **jedz()** i **halasuj()** — dzięki czemu wszystkie typy zwierząt będą mogły zdefiniować swój własny sposób pożywiania się, i wydawania odgłosów. Jak na razie wydaje się, że metody **spij()** i **wedruj()** mogą być zdefiniowane ogólnie dla wszystkich typów zwierząt.

- 3 Określ, czy klasy potomne muszą mieć zachowania (implementacje metod) charakterystyczne dla swojego typu.

Po przeanalizowaniu klasy **Zwierze** doszliśmy do wniosku, że w poszczególnych klasach potomnych należy przesłonić metody **jedz()** oraz **halasuj()**.

Poszukiwanie dalszych możliwości zastosowania dziedziczenia

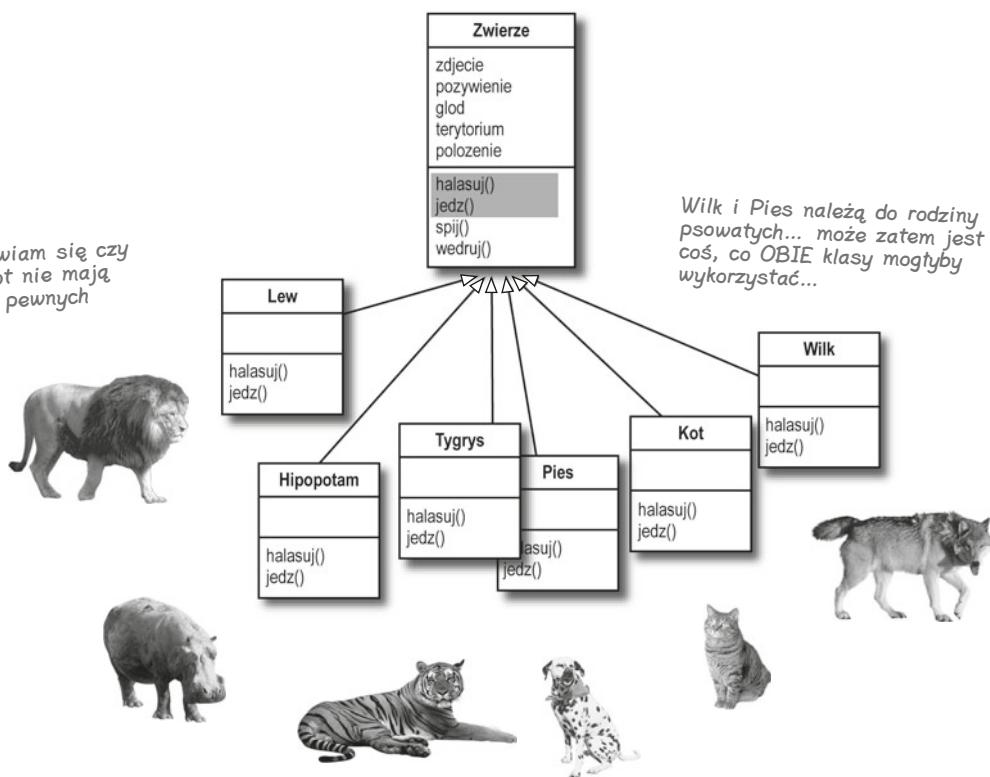
Powoli zaczyna się wykształtać hierarchia tworzących klas. Każda z klas potomnych przesyła metody jedz() i halasuj(), dzięki czemu nie będzie już można pomylić szczekania Psa i miauczenia Kota (co byłoby bardzo obraźliwe dla obu tych gatunków), a Hipopotam nie będzie się pozywał tak samo jak Lew.

Ale może można zrobić jeszcze więcej. Musimy przeanalizować klasy potomne klasy Zwierze i spróbować określić, czy w jakiś sposób można połączyć dwie lub kilka z nich i wydzielić pewien kod unikalny wyłącznie dla *tej* nowej grupy klas. Podobieństwa wykazują klasy Wilk i Pies oraz Lew, Tygrys i Kot.

- 4 Poszukaj dalszych możliwości określenia kolejnych poziomów abstrakcji – wyszukaj klasy potomne, które mogą mieć wspólne zachowania.

Przeanalizowaliśmy nasze klasy i zauważliśmy, że wspólne zachowania mogą mieć klasy Wilk i Pies oraz Lew, Tygrys i Kot.

Hmm... Zastanawiam się czy Lew, Tygrys i Kot nie mają przez przypadek pewnych wspólnych cech.

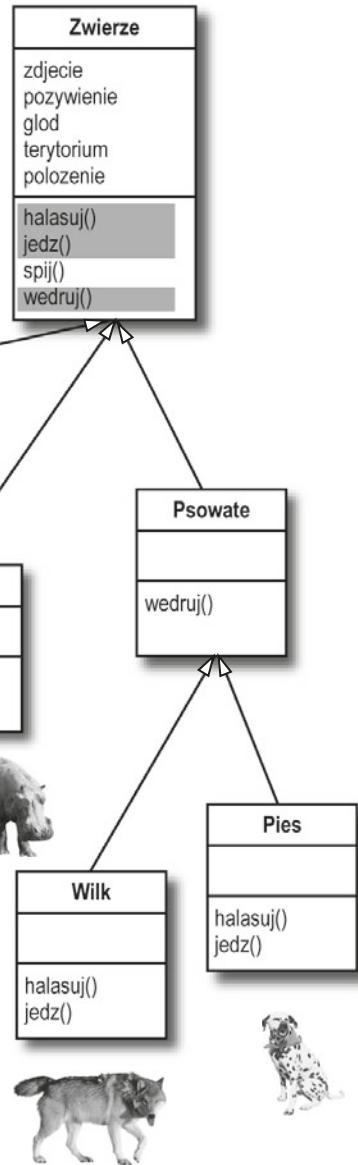


5 Dokończenie hierarchii klas

W biologii została już określona hierarchia i organizacja zwierząt (królestwa, rodzaje, gromady itd.), dlatego możemy wykorzystać pewne elementy tej hierarchii, które mają sens w odniesieniu do naszego projektu klas. Do dalszej organizacji zwierząt wykorzystamy biologiczne rodziny, którym będą odpowiadać klasy Kotowate i Psowate.

Zdecydowaliśmy, że Psowate mogłyby korzystać ze wspólnej metody `wedruij()`, gdyż zazwyczaj zwierzęta te wędrują gromadnie. Zauważliśmy, że także Kotowate mogą używać tej samej wspólnej metody `wedruij()`, gdyż zazwyczaj starają się one unikać innych osobników tego samego gatunku. Klasa Hipopotam wciąż będzie korzystać z odziedziczonej metody `wedruij()` zdefiniowanej w klasie Zwierze.

A zatem na razie skończyliśmy z naszym projektem — wrócimy do niego ponownie w dalszej części rozdziału.



Jaka metoda jest wywoływana?

Klasa Wilk ma cztery metody. Jedną odziedziczoną po klasie Zwierze, jedną po klasie Psowate (która w rzeczywistości jest przesłoniętą wersją metody z klasy Zwierze) oraz dwie metody przesłonięte bezpośrednio w klasie Wilk. Jeśli utworzysz obiekt klasy Wilk i zapiszesz go w zmiennej, będziesz mógł używać tej zmiennej wraz z operatorem kropki do wywoływania wszystkich czterech dostępnych metod. Jednak które *wersje* tych metod zostaną wywołane?

utworzenie nowego obiektu Wilk

```
Wilk w = new Wilk();
```

wywołuje metodę klasy Wilk

```
w.halasuj();
```

wywołuje metodę klasy Psowate

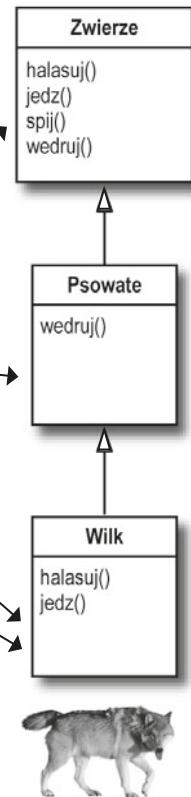
```
w.wedruj();
```

wywołuje metodę klasy Wilk

```
w.jedz();
```

wywołuje metodę klasy Zwierze

```
w.spij();
```



Kiedy, posługując się odwołaniem do obiektu, wywołujesz pewną metodę, to zostanie wywołana najbardziej szczegółowa wersja tej metody dostępna w danym obiekcie.

Innymi słowy: *najniższa wygrywa!*

„Najniższa” odnosi się w tym przypadku do położenia w drzewie hierarchii klas. Klasa Psowate jest położona poniżej klasy Zwierze, a klasa Wilk poniżej klasy Psowate, dlatego też w przypadku użycia odwołania do obiektu Wilk do wywołania jakieś metody, JVM w pierwszej kolejności sprawdzi klasę Wilk. Jeśli poszukiwana metoda nie zostanie w niej odnaleziona, to wirtualna maszyna Javy będzie przechodzić do kolejnych klas zajmujących wyższe miejsca w drzewie hierarchii aż do momentu odnalezienia metody.

Ćwiczenia z projektowania drzewa dziedziczenia

Projektowanie drzewa dziedziczenia

Klasa	Klasa bazowa	Klasa potomna
Ubranie	---	Bokserki, Koszulka
Bokserki	Ubranie	
Koszulka	Ubranie	

Tabela dziedziczenia

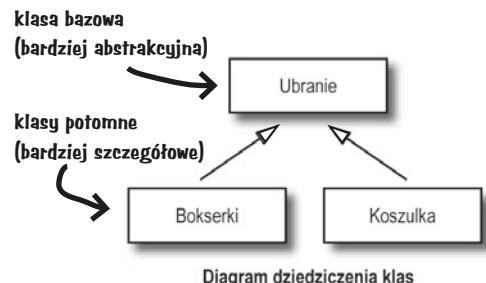


Tabela dziedziczenia

Diagram dziedziczenia klas



Zaostrz ołówek

Tutaj narysuj diagram hierarchii klas.

Znajdź sensowne relacje. Wypełnij dwie ostatnie kolumny.

Klasa	Klasa bazowa	Klasa potomna
Muzyk		
Gwiazda Rocka		
Fan		
Basista		
Pianista Koncertowy		

Podpowiedź: Nie wszystkie klasy można ze sobą połączyć.

Podpowiedź: Możesz dodać nowe lub zmienić klasy podane w tabeli.

Nie istnieja
glupie pytania

P: Napisaliście, że JVM rozpoczęła sprawdzanie drzewa dziedziczenia od klasy, dla jakiej metoda została wywołana (tak było w przypadku klasy Wilk we wcześniejszym przykładzie). Jednak co się stanie, jeśli wirtualna maszyna Javy w ogóle nie znajdzie poszukiwanej metody?

O: To dobre pytanie! Na szczęście nie musisz się o to martwić. Kompilator gwarantuje, że dla odwołania danego typu można będzie wywołać konkretną metodę, choć nie określa, do jakiej klasy metoda ta będzie należeć podczas działania programu (ani go to w ogóle nie interesuje). W przykładzie z klasą Wilk kompilator poszukuje metody spij(), lecz wcale nie interesuje go fakt, że metoda ta jest zdefiniowana w klasie Zwierze

(i dziedziczonej po niej). Pamiętaj, że jeśli klasa dziedziczy metodę, to po prostu dysponuje tą metodą. To, gdzie metoda została zdefiniowana (innymi słowy, w której z klas bazowych), nie ma dla kompilatora najmniejszego znaczenia. Jednak podczas działania programu wirtualna maszyna Javy zawsze wybierze odpowiednią metodę. „Odpowiednią” czyli najbardziej szczegółową spośród metod dostępnych dla danego typu obiektu.

Relacje JEST oraz MA

Pamiętaj, że gdy jedna klasa dziedziczy po drugiej, to mówimy, że klasa potomna *rozszerza* klasę bazową. Kiedy chcesz się dowiedzieć, czy jedna rzecz powinna rozszerzać inną, zastosuj test relacji JEST.

Trójkąt JEST figurą — o tak, to prawda.

Chirurg JEST lekarzem — to też prawda.

Wanna rozszerza łazienkę... — cóż, w pewnym sensie brzmi to sensownie.

Jednak tylko do momentu, gdy zastosujemy test relacji JEST.

Jeśli chcesz określić, czy dobrze zaprojektowałeś swoje typy danych, zadaj sobie pytanie: „Czy stwierdzenie, że typ X JEST typem Y ma sens?”. Jeśli odpowiedź na to pytanie jest przecząca, oznacza to, że w projekcie klas coś jest nie tak. A zatem, jeśli zastosujemy powyższy test, okaże się, że odpowiedź na pytanie „czy Wanna JEST Lazienka” jest przecząca.

A co się stanie, jeśli odwróciśmy sytuację i zastanowimy się, czy Lazienka rozszerza Wannę? Także ten wariant nie jest dobry, gdyż to nie prawda, że Lazienka JEST Wanna.

Zarówno Lazienka, jak i Wanna są ze sobą skojarzone, jednak nie łączy ich relacja dziedziczenia. Obie te klasy łączy relacja MA. Czy stwierdzenie „Lazienka MA Wanne” ma sens? Jeśli tak, to będzie to oznaczać, że obiekt Lazienka zawiera składową typu Wanna. Innymi słowy, Lazienka zawiera odwołanie do Wanny, lecz ani Wanna nie rozszerza Lazienki, ani Lazienka nie rozszerza wannę.



Lazienka MA Wanne, a Wanna MA Babelki. Jednak żadna klasa nie dziedziczy po żadnej innej (czyli nie rozszerza jej).

Czy stwierdzenie Wanna JEST Lazienka ma sens? Cóż, dla mnie nie ma. Relacja występująca pomiędzy klasami Wanna i Lazienka jest relacją typu MA. Lazienka MA Wannę. Co oznacza, że Lazienka zawiera składową typu Wanna.



Ale poczekaj! To jeszcze nie wszystko!

Test relacji JEST działa w *dowolnym* miejscu drzewa dziedziczenia. Jeśli drzewo dziedziczenia zostało dobrze zaprojektowane, to test ten powinien być spełniony zawsze, gdy zadamy pytanie, czy *dowolna* klasa potomka JEST *dowolną* z jej klas bazowych.

Jeśli klasa B rozszerza klasę A, to klasa B JEST kąsa A.

Stwierdzenie to jest prawdziwe niezależnie od położenia klas w drzewie dziedziczenia. Jeśli klasa C rozszerza klasę B, to test relacji JEST będzie spełniony zarówno dla klas C i B, jak i dla klas C i A.

Klasa Psowate rozszerza klasę Zwierze

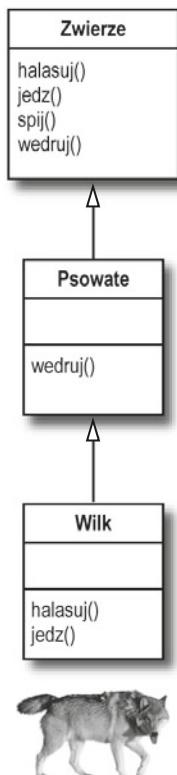
Klasa Wilk rozszerza klasę Psowate

Klasa Wilk rozszerza klasę Zwierze

Psowate SĄ klasą Zwierze

Wilk JEST klasą Psowate

Wilk JEST klasą Zwierze



Jeśli dysponujemy drzewem dziedziczenia takim jak to przedstawione obok, zawsze będziemy mogli stwierdzić, że: „**klasa Wilk rozszerza klasę Zwierze**” lub „**Wilk JEST klasą Zwierze**”. Fakt, że Zwierze jest klasą bazową klasy bazowej klasy Wilk, nie ma żadnego znaczenia. W rzeczywistości o ile tylko klasa Zwierze będzie istnieć w hierarchii dziedziczenia *gdziekolwiek* powyżej klasy Wilk, to relacja Wilk JEST klasą Zwierze zawsze będzie prawdziwa.

Struktura drzewa hierarchii naszych klas reprezentujących zwierzęta informuje, że: „**Wilk JEST klasą Psowate**, zatem może zrobić dokładnie to samo co Psowate. Co więcej, **Wilk JEST klasą Zwierze**, zatem może robić wszystko to co każde Zwierze.”

Nawet jeśli Wilk przesłania pewne metody klas Zwierze lub Psowate, nie ma to najmniejszego znaczenia. Z punktu widzenia całego świata (innego kodu) Wilk udostępnia cztery metody. W jaki sposób metody te są wykonywane lub w jakiej klasie zostały one przesłonięte nie ma znaczenia. Wilk potrafi hałaśować, jeść, spać i wędrować gdyż dziedziczy po klasie Zwierze.



Jak możesz określić, czy dobrze zaprojektowałeś hierarchię dziedziczenia?

Oczywiście nie wyczerpalismy poruszanej problematyki, jednak znacznie więcej zagadnień związanych z programowaniem zorientowanym obiektywo opiszemy w następnym rozdziale (gdzie wprowadzimy ostateczne zmiany i poprawki w projekcie przedstawionym w tym rozdziale).

Jak na razie dobrym rozwiązaniem będzie korzystanie z testu relacji JEST. Jeśli stwierdzenie „X JEST Y” ma sens, to obie klasy (X oraz Y) prawdopodobnie powinny znaleźć się w jednej hierarchii dziedziczenia. Być może istnieje szansa, że obie klasy będą miały jakieś powielające się zachowania.

Pamiętaj, że relacja JEST występująca w przypadku dziedziczenia działa tylko w jednym kierunku!

Stwierdzenie, że Trojkat jest klasą Figura ma sens, zatem klasa Trojkat może rozszerzać klasę Figura.

Jednak odwrócenie relacji i stwierdzenie, że Figura JEST klasą Trojkat *nie* ma sensu i dlatego klasa Figura nie powinna rozszerzać klasę Trojkat. Pamiętaj, że relacja JEST oznacza, że klasa X JEST klasą Y, zatem X jest w stanie zrobić to samo co Y (a może i więcej).

Zaostrz ołówek

Zaznacz, które z poniższych relacji są sensowne:

- Piekarnik** rozszerza **Kuchnię**
- Gitaro** rozszerza **Instrument**
- Osoba** rozszerza **Pracownika**
- Ferrari** rozszerza **Silnik**
- JajkoSadzone** rozszerza **Pożywienie**
- Hart** rozszerza **ZwierzątkoDomowe**
- Pojemnik** rozszerza **Słoik**
- Metal** rozszerza **Tytan**
- Papa dance** rozszerza **Kapela**
- Blondynka** rozszerza **Mądrala**
- Napój** rozszerza **Martini**

Podpowiedź: Zastosuj test JEST.

| Nie istnieja
główne pytania

P: Zatem wiemy, jak klasa potomna uzyskuje dostęp do dziedziczonych metod klasy bazowej. A co się dzieje, jeśli to klasa bazowa chciałaby użyć wersji metody zdefiniowanej w klasie potomnej?

O: Otóż klasa bazowa wcale niekoniecznie musi wiedzieć o istnieniu klas potomnych. Możesz stworzyć klasę, a ktoś zupełnie inny, dużo później, może napisać jej klasę potomną. Jednak nawet jeśli twórcą klasy bazowej wiedziałby o istnieniu nowej wersji metody zdefiniowanej w klasie potomnej (i chciał jej użyć), to i tak nie istnieje żaden mechanizm, który by mu na to pozwolił (coś, co można by określić mianem *odwrotnego dziedziczenia*). Zastanów się, to dzieci dziedziczą po swoich rodzicach, a nie na odwrót.

P: A co, jeśli w klasie potomnej chciałbym użyć zarówno metody zdefiniowanej w klasie bazowej, jak i jej przesiąkniętej wersji zdefiniowanej w danej klasie? Innymi słowy, jeśli nie chcę całkowicie zastępować metody klasy bazowej, a jedynie ją rozbudować.

O: To można zrobić! I jest to bardzo ważny czynnik projektowania klas. Wyobraź sobie, że słowo „rozszerza” oznacza „rozszerzenie możliwości funkcjonalnych klasy bazowej”.

public void wedruj () {

super.wedruj ();
// mój własny kod związany
// z wędrowaniem
}

Możesz zaprojektować metody klasy bazowej w taki sposób, aby ich implementacja mogła działać we wszystkich klasach potomnych, nawet jeśli w tych klasach trzeba będzie „dodać” do nich nowe fragmenty kodu. Aby w metodzie klasy potomnej wywołać przesłanianą metodę klasy bazowej, należy użyć słowa kluczowego **super**. To tak, jakby powiedział: „najpierw wykonaj wersję metody zdefiniowaną w klasie bazowej, a potem wróć i wykonaj mój kod...”.

ta instrukcja wywołuje dziedziczoną wersję metody wedruj(), a potem wracamy, aby wykonać własny kod.

Kto dostanie Porshe, a kto model z plasteliny? (czyli jak można się dowiedzieć, co klasa potomna może dziedziczyć po klasie bazowej)

Klasa potomna dziedziczy zarówno składowe, jak i metody klasy bazowej, choć w dalszej części książki przedstawimy jeszcze inne dziedziczone elementy klas. Klasa bazowa może zdecydować, czy chce, aby klasa potomna dziedziczyła pewną składową lub metodę, określając jej poziom dostępu.



Istnieją cztery poziomy dostępu i wszystkie je opiszymy w niniejszej książce. Oto one, zaczynając od poziomu, który narzuca największe ograniczenia:

prywatny domyślny chroniony publiczny

Poziomy dostępu określają, kto i co może widzieć.

Stanowią one jeden z kluczowych czynników określających, czy kod Javy jest dobrze zaprojektowany i solidny. Jak na razie skoncentrujemy się jedynie na dostępie publicznym i prywatnym. W tych dwóch przypadkach zasady są bardzo proste:

składowe i metody publiczne są dziedziczone

składowe i metody prywatne nie są dziedziczone

Jeśli klasa potomna dziedziczy składową lub metodę, *to w zasadzie można je potraktować jako składowe i metody zdefiniowane bezpośrednio w tej klasie potomnej*.

W przedstawionym wcześniej przykładzie z klasą Figura klasa Kwadrat dziedziczyła metodę obroc() oraz odtworzDzwiek(), jednak dla świata zewnętrznego (czyli innego kodu) klasa Kwadrat po prostu miała obie te metody. A zatem składowe klas obejmują składowe zdefiniowane w danej klasie oraz wszystkie składowe dziedziczone po klasach bazowych; podobnie jest z metodami.

Więcej szczegółowych informacji o odstępie domyślnym i chronionym znajdziesz rozdziale 17. (w części poświęconej wdrażaniu) oraz w dodatku B.

Czy korzystanie z dziedziczenia przy projektowaniu klas jest „używaniem”, czy „nadużywaniem”?

Choć przyczyny uzasadniające niektóre z podanych poniżej zasad zostaną wyjaśnione dopiero w dalszych częściach książki, to jak na razie sama znajomość tych zasad pomoże Ci projektować lepsze klasy wykorzystujące dziedziczenie.

UŻYWAJ dziedziczenia, gdy jedna z klas jest bardziej szczegółowym typem klasy bazowej. Na przykład Brzoza jest szczególnym typem Drzewa, zatem ma sens, aby klasa Brzoza *rozszerzała* klasę Drzewo.

UŻYWAJ dziedziczenia w sytuacjach, gdy dysponujesz operacjami (czyli zaimplementowanym kodem), które powinny być współużytkowane przez wiele klas tego samego typu ogólnego. Na przykład każda z klas Kwadrat, Trojkąt i Okrąg musi się obracać i odtwarzać dźwięki; dlatego też umieszczenie tych możliwości funkcjonalnych w klasie bazowej Figura ma sens i dodatkowo ułatwia pielęgnację i rozbudowę kodu. Pamiętaj jednak, że choć dziedziczenie jest jedną z podstawowych cech programowania obiektowego, to jednak nie zawsze jest optymalnym sposobem wielokrotnego stosowania pewnych zachowań i możliwości klas. Bez wątpienia pomoże Ci na początku i niejednokrotnie jest właściwym rozwiązaniem, jednak trzeba także pamiętać o wzorcach projektowych, które udostępniają wiele innych, znacznie subtelniejszych i bardziej elastycznych opcji. Jeśli jeszcze nie wiesz nic o wzorcach projektowych, to doskonałym uzupełnieniem tej lektury będzie książka *Wzorce projektowe. Rusz głową!*

NIE UŻYWAJ dziedziczenia tylko i wyłącznie po to, aby ponownie wykorzystać kod zdefiniowany w innej klasie, jeśli relacja pomiędzy klasą bazową a potomną narusza któryś z powyższych reguł. Na przykład wyobraź sobie, że w klasie Alarm zaimplementowałeś specjalny kod drukujący, a teraz potrzebujesz podobnego kodu w klasie Pianino. Cóż zatem robisz? Definiujesz klasę Pianino w taki sposób, aby rozszerzała klasę Alarm i dziedziczyła potrzebny kod. Ale to przecież nie ma sensu! Pianino *nie* jest żadnym szczególnym typem Alarmu. (A zatem kod drukujący powinien być zdefiniowany w klasie Drukarka, tak aby wszystkie klasy, które muszą coś drukować, mogły z niej skorzystać, posługując się relacją MA).

NIE UŻYWAJ dziedziczenia, jeśli klasa potomna i bazowa nie spełniają testu relacji JEST. Zawsze się zastanów, czy klasa potomna jest bardziej wyspecjalizowanym lub szczególnym typem klasy bazowej. Na przykład stwierdzenie „Herbata JEST Napojem” ma sens. Jednak stwierdzenie odwrotne — „Napój JEST Herbatą” — nie ma sensu.



CELNE SPOSTRZEŻENIA

- Klasa potomna *rozszerza* klasę bazową.
- Klasa potomna dziedziczy wszystkie *publiczne* składowe i metody klasy bazowej, jednak nie dziedziczy składowych i metod *prywatnych*.
- Metody odziedziczone *można* przesłaniać, z kolei przesłanianie odziedziczonych składowych *nie jest możliwe* (można natomiast te składowe *ponownie zdefiniować*, choć nie jest to to samo i niemal nigdy nie ma potrzeby stosowania takiego rozwiązania).
- Zastosuj test relacji JEST, aby sprawdzić, czy stworzony projekt hierarchii jest poprawny. Jeśli X *rozszerza* Y, to stwierdzenie, że X JESTY musi mieć sens.
- Relacja JEST działa tylko w jednym kierunku. Hipopotam jest Zwierzęciem, jednak nie wszystkie Zwierzęta są Hipopotamami.
- Jeśli pewna metoda zostanie przesłonięta w klasie potomnej i jeśli wywołamy ją posługując się obiektem tej klasy potomnej, to zostanie wykonana metoda zdefiniowana w klasie potomnej (zgodnie z zasadą — „*najniższa wygrywa*”).
- Jeśli klasa B rozszerza klasę A, a klasa C rozszerza klasę B, to klasa B JEST klasą A, klasa C JEST klasą B, a jednocześnie klasa C JEST klasą A.

Co w rzeczywistości daje nam dziedziczenie?

Wiele zyskujesz, projektując swoje klasy w taki sposób, aby wykorzystywały możliwości, jakie daje dziedziczenie. Wydzielając zachowania wspólne dla grupy klas i umieszczając je w klasie bazowej, możesz pozbyć się powielanego kodu. Dzięki temu w razie konieczności zaktualizowania tego kodu, zmiany będziesz musiał wprowadzić tylko w jednym miejscu, a *w jakiś magiczny sposób zostaną one zauważone we wszystkich klasach, które to zachowanie dziedziczą*. No dobrze, tak naprawdę nie ma w tym żadnej magii, a wszystko jest bardzo proste — wystarczy wprowadzić modyfikacje i skompilować klasę. To wszystko.

W żaden sposób nie trzeba modyfikować klas potomnych!

Wystarczy udostępnić zmodyfikowaną klasę bazową, a wszystkie klasy, które ją rozszerzają, automatycznie wykorzystają jej nową wersję.

Program napisany w Javie to właściwie grupa klas, dzięki czemu, aby użyć nowej wersji klasy bazowej, nie trzeba komplikować klas potomnych. O ile tylko klasa bazowa nie powoduje żadnych problemów w klasach potomnych, wszystko jest w porządku. (W dalszej części książki bardziej szczegółowo opiszymy te „problemy”. Jak na razie wyobraź sobie, że są one spowodowane zmieniem w klasie bazowej czegoś, od czego zależy działanie klas potomnych, na przykład argumentów metody, typu zwracanej wartości, nazwy metody itp.)

① Unikasz powielania kodu.

Umieść kod używany w różnych klasach, w jednym miejscu i pozwól, aby klasy potomne dziedziczyły go po klasie bazowej. Chcąc zmienić dane zachowanie, będziesz musiał wprowadzić modyfikacje tylko w jednym miejscu, a wszystkie klasy potomne zauważą zmiany.

② Zdefiniuj wspólny protokół dla grupy klas.



Dzięki dziedziczeniu możesz zagwarantować, że wszystkie klasy potomne będą mieć wszystkie metody, które ma klasa bazowa*

Innymi słowy, definiujesz wspólny protokół, który będzie posiadać grupa klas związanych ze sobą relacją dziedziczenia.

Definiując w klasie bazowej metody, które mogą być dziedziczone przez klasy potomne, tworzysz pewien protokół, który informuje pozostałe części programu, że: „wszystkie moje podtypy (czyli klasy potomne) potrafią wykonywać te operacje, wykorzystując do tego celu następujące metody...”.

Innymi słowy, tworzysz pewien *kontrakt*.

Klasa *Zwierze* definiuje protokół wspólny dla wszystkich pozostałych typów zwierząt:

Zwierze
halasuj() jedz() spij() wedruj()

Informujesz cały świat, że dowolne Zwierze może wykonywać cztery poniższe czynności. Dotyczy to także argumentów i wartości wynikowych metod.

I pamiętaj, że mówiąc *dowolne Zwierze* mamy na myśli zarówno klasę *Zwierze*, jak i *wszystkie jej klasy potomne*. A to oznacza dowolną klasę, która w drzewie hierarchii dziedziczenia jest położona poniżej klasy *Zwierze*.

Jednak nie doszliśmy jeszcze do tego, co jest najlepsze, bowiem najlepszą zabawę — *polimorfizm* — zostawiliśmy na sam koniec.

Definiując typ bazowy dla pewnej grupy klas, *w dowolnym miejscu, gdzie jest oczekiwana klasa bazowa, można umieścić dowolną z klas potomnych*.

Co proszę?

Nie przejmuj się, dopiero będziemy to wyjaśniać.

Po przeczytaniu dwóch kolejnych stron będziesz już ekspertem.

Zwracam na to dużą uwagę...

gdź zaczniesz wykorzystać możliwości, jakie daje polimorfizm.

To ma dla mnie znaczenie...

gdź zaczniesz odwoływać się do obiektów klas potomnych, wykorzystując odwołania typu bazowego.

A to oznacza, że...

zaczniesz pisać naprawdę elastyczny kod. Kod, który jest znacznie bardziej przejrzysty (bardziej efektywny i prostszy). Kod, który nie tylko jest łatwiej tworzyć, lecz także rozbudowywać i to na sposoby, o których w ogóle nie myślałeś, pisząc ten kod.

A to oznacza, że możesz pojechać na wakacje na tropikalną wyspę, kiedy Twoi współpracownicy będą aktualizować program, a co ciekawsze, nawet nie będą musieli mieć dostępu do Twojego kodu źródłowego.

Na następnej stronie zobaczysz, jak to wszystko działa.

Nie znamy Twoich preferencji wakacyjnych, jednak w naszym przypadku perspektywa wakacji na tropikalnej wyspie jest bardzo motywująca.



* Mówiąc „wszystkie metody”, mamy na myśli wszystkie te metody, które mogą być dziedziczone. Aktualnie oznacza to „wszystkie metody publiczne”, choć w dalszej części książki nieco zmienimy tę definicję.

Trzy etapy deklarowania i przypisywania obiektu

1 3 2
Pies mojPies = new Pies();

Aby się przekonać, jak działa polimorfizm, musimy się nieco cofnąć i przypomnieć sobie normalny sposób, w jaki deklarowane są odwołania i tworzone obiekty...

1 Deklaracja zmiennej referencyjnej

Pies mojPies = new Pies();

Informuje wirtualną maszynę Javy, że należy przydzielić pamięć dla zmiennej referencyjnej. Zmienna ta, już na zawsze, będzie typu Pies. Innymi słowy będzie ona pilotem posiadającym przyciski umożliwiające sterowanie Psem, ale nie Kotem ani obiektem Socket.



2 Utworzenie obiektu

Pies mojPies = new Pies();

Nakazuje wirtualnej maszynie Javy przydzielić na stercie obszar przeznaczony na nowy obiekt klasy Pies.

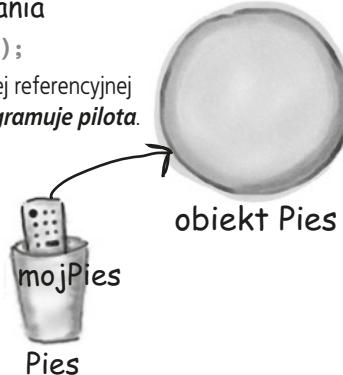


obiekt Pies

3 Połączenie obiektu i odwołania

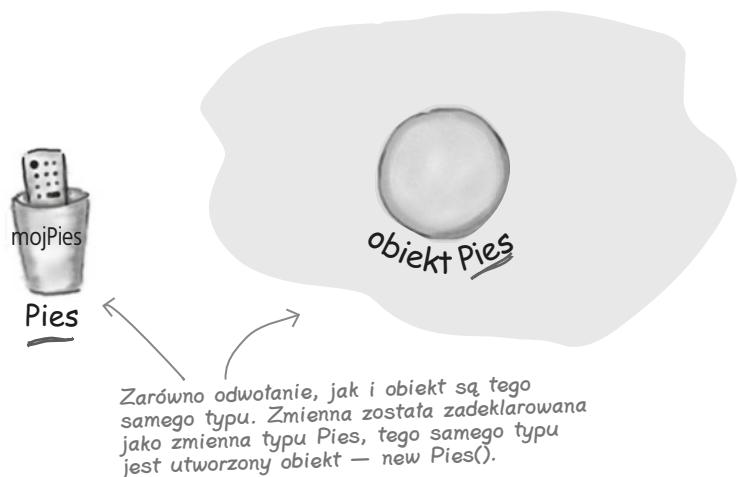
Pies mojPies = new Pies();

Zapisuje obiekt Pies w zmiennej referencyjnej mojPies. Innymi słowy — *programuje pilota*.



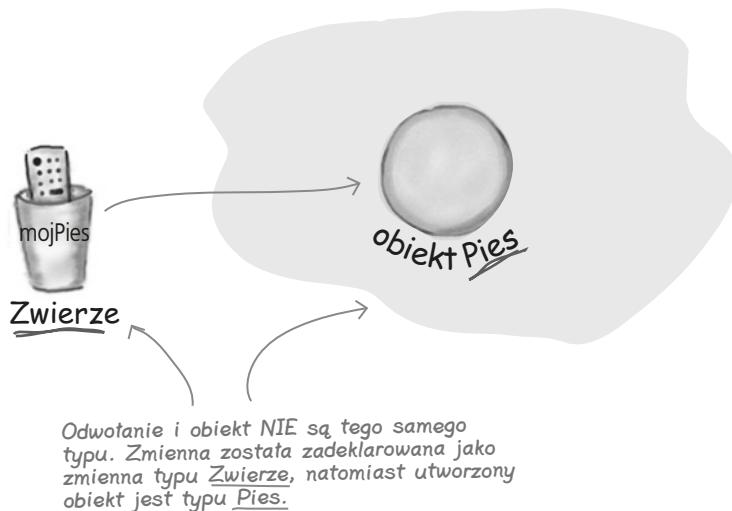
Najważniejsze jest to, że typ odwołania i obiekt są takie same.

W naszym przykładzie, odwołanie i obiekt są typu Pies.



Jednak dzięki zastosowaniu polimorfizmu zarówno obiekt, jak i odwołanie do niego mogą być różnych typów.

Zwierze mojPies = new Pies();



Polimorfizm w działaniu

Dzięki polimorfizmowi typ odwołania nie musi być zgodny z typem faktycznego obiektu — podczas definiowania odwołania można użyć typu bazowego.

Po zadeklarowaniu zmiennej referencyjnej pewnego typu można w niej zapisać obiekt dowolnego typu, który spełnia test relacji JEST z zadeklarowanym typem zmiennej. Innymi słowy, w zmiennej można zapisać referencję do obiektu dowolnego typu, który rozszerza jej zadeklarowany typ. *To pozwala nam tworzyć, na przykład, tablice polimorficzne.*



No dobrze... Może przykład coś wyjaśni.

```
Zwierze[] zwierzeta = new Zwierzeta[5];
```

Deklarujemy tablicę typu *Zwierze*.
Innymi słowy tworzymy tablicę, która
będzie zawierać obiekty typu *Zwierze*.

```
zwierzeta[0] = new Pies();
```

Ale spójrz na to. W tablicy typu
Zwierze można zapisywać obiekty
DOWOLNYCH klas potomnych
klasy *Zwierze*!

```
zwierzeta[1] = new Kot();
```

A to jest najciekawszy aspekt polimorfizmu
(a zarazem powód przedstawienia tego
przykładu) — przeglądamy całą tablicę
i wywołujemy jedną z metod klasy *Zwierze*,
a wszystkie obiekty wykonują właściwe
czynności!

```
zwierzeta[2] = new Wilk();
```

```
zwierzeta[3] = new Hipopotam();
```

```
zwierzeta[4] = new Lew();
```

```
for (int i = 0; i < zwierzeta.length; i++) {
```

Kiedy zmienna „i” ma wartość 0, w komórce
tablicy o tym indeksie znajduje się obiekt
Pies, a zatem, wywołana zostanie metoda
jedz() klasy *Pies*. Kiedy zmienna „i” przyjmie
wartość 1, zostanie wywołana metoda *jedz()*
klasy *Kot*.

```
    zwierzeta[i].jedz();
```

To samo dotyczy metody *wedruj()*.

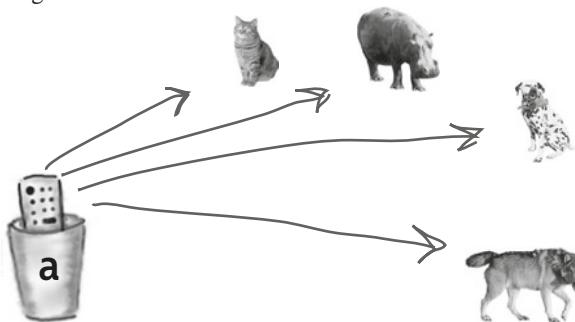
```
    zwierzeta[i].wedruj();
```

```
}
```

Ale poczekaj! To nie wszystko!

Można używać polimorficznych typów argumentów i wartości wynikowych.

Można zadeklarować zmienną referencyjną typu bazowego, na przykład `Zwierze`, a następnie zapisać w niej obiekt typu potomnego, na przykład `Pies`. Pomyśl, jakie to daje możliwości w sytuacji, gdy odwołanie jest przekazywane jako argument wywołania metody...



```
class Weterynarz {

    public void zrobZastrzyk(Zwierze a) {
        // straszne przeżycia dla zwierzęcia
        // określonego parametrem "a"
        a.halasuj();
    }
}
```

```
class Wlasciciel {

    public void start() {
        Weterynarz w = new Weterynarz();
        Pies p = new Pies(); ←
        Hipopotam h = new Hipopotam(); ←
        w.zrobZastrzyk(p); ← Wykonana zostanie metoda
        w.zrobZastrzyk(h); ← Wykonana zostanie metoda
    }
}
```

W parametrze 'a' można zapisać argument DOWOLNEGO typu `Zwierze`. A kiedy `Weterynarz` skończy robienie zastrzyku i każe zwierzęciu wydać jakiś odgłos, to zostanie wywołana metoda `halasuj()` klasy, której obiekt zostanie przekazany jako argument.

Do metody `zrobZastrzyk()` klasy `Weterynarz` można przekazać obiekt dowolnej klasy `Zwierze`. Wszystko będzie w porządku, o ile typ obiektu przekazanego jako argument będzie dowolną klasą potomną klasy `Zwierze`.

Wykonana zostanie metoda `halasuj()` klasy `Pies`.

Wykonana zostanie metoda `halasuj()` klasy `Hipopotam`.

Stosowanie możliwości polimorfizmu



TERAZ już rozumiem! Jeśli napiszę swój kod, wykorzystując argumenty polimorficzne, czyli argumenty klasy bazowej, to potem, podczas działania programu, będę mogła przekazywać do metody obiekty dowolnej klasy potomnej. Ale super! Oznacza to bowiem, że mogę napisać swój kod, pojechać na wakacje, a kiedy ktoś inny doda do programu nowe klasy potomne, to moje metody i tak będą działać... (szkoda tylko, że utatwi to życie temu durniowi Mirkowi).

Dzięki polimorfizmowi można tworzyć kod, którego nie trzeba będzie zmieniać w przypadku dodawania do programu nowych klas potomnych.

Pamiętasz jeszcze klasę `Weterynarz`? Jeśli stworzysz tę klasę, używając argumentów typu `Zwierze`, to Twój kod będzie w stanie obsługiwać dowolne *klasy potomne* klasy `Zwierze`. To oznacza, że inni programiści, chcąc użyć Twojej klasy `Weterynarz`, będą jedynie musieli upewnić się, że *ich* nowe typy zwierząt rozszerzają klasę `Zwierze`. Metody klasy `Weterynarz` będą działać, choć w czasie jej tworzenia nie były znane klasy potomne klasy `Zwierze`, na których `Weterynarz` będzie operować.



Dlaczego mamy gwarancję, że polimorfizm będzie działać w ten sposób? Dlaczego możemy bezpiecznie założyć, że dowolna *klasa potomna* będzie mieć metody, które wywołujemy dla obiektów jej *klasy bazowej* (używając odwołań do klasy bazowej i operatora kropki)?

Nieistniejąca grupa pytań

P: Czy istnieje jakieś praktyczne ograniczenie ilości poziomów klas potomnych? Jak „głęboko” możemy jeździć?

O: Jeśli przejrzesz strukturę Java API, to zapewne zauważysz, że większość hierarchii dziedziczenia jest rozbudowana w poziomie, ale nie w pionie. Większość z nich ma tylko jeden lub dwa poziomy „głębokości”, choć oczywiście istnieją wyjątki (głównie chodzi tu o klasy związane z graficznym interfejsem użytkownika). Kiedyś sam dojdzieś do wniosku, że najlepszym rozwiązańiem jest tworzenie niezbyt „głębokich” hierarchii dziedziczenia, choć nie ma żadnego stałego ograniczenia (a przynajmniej takiego, które mógłbyś przekroczyć).

P: Hej, właśnie coś wpadło mi do głowy... Jeśli nie masz dostępu do kodu źródłowego klasy, a chcesz zmienić sposób działania którejś z jej metod, to czy możesz w tym celu stworzyć klasę potomną, aby rozszerzyć „złą” klasę i przesłonić metody, które chcesz zastąpić swoim własnym, lepszym kodem?

O: Jasne, że tak! To jedna z najlepszych możliwości programowania obiektowego, która dodatkowo może uchronić Cię od konieczności pisania całego kodu klasy od początku lub ścigania programisty, który go stworzył.

P: Czy można rozszerzać wszystkie klasy? Czy też, podobnie jak jest w przypadku składowych, nie można rozszerzać klas, które zostały zadeklarowane jako prywatne?

O: Nie ma czego takiego jak „klasa prywatna”; no może za wyjątkiem bardzo szczególnych przypadków — tak zwanych *klas wewnętrznych* — o których w ogóle jeszcze nie mówiliśmy. Niemniej jednak istnieją trzy czynniki, które mogą uniemożliwić tworzenie klas potomnych pewnej klasy.

Pierwszym z nich jest kontrola dostępu. Choć klasa *nie może* być oznaczona jako prywatna, to jednak może być niepubliczna (co można uzyskać poprzez pominięcie słowa kluczowego `public` w deklaracji klasy). W takim przypadku klasy potomne mogą tworzyć wyłącznie w obrębie tego samego pakietu, do którego należy rozszerzana klasa. Klasy należące do innych pakietów nie będą mogły jej rozszerzać (właściwie to nawet nie będą mogły jej *używać*).

Kolejnym czynnikiem, który może uniemożliwić tworzenie klas potomnych, jest modyfikator `final`. Klasa finalna oznacza „koniec linii dziedziczenia”. Nikt i nigdy nie może rozszerzać klas finalnych.

I w końcu, jeśli klasa ma wyłącznie prywatne konstruktory (konstruktorami zajmiemy się w rozdziale 9.), to także nie można tworzyć jej klas potomnych.

P: Po co w ogóle tworzyć klasy finalne? Jaki korzyści może dawać uniemożliwienie tworzenia klas potomnych?

O: Zazwyczaj nie będziesz tworzyć finalnych klas. Jeśli jednak potrzebujesz poczucia bezpieczeństwa — poczucia wynikającego z pewności, że Twoje metody działają dokładnie tak, jak je napisałeś (gdyż nie można ich przesłonić) — to właśnie klasa finalna może Ci je zapewnić. Właśnie z tego powodu w Java API można znaleźć wiele klas finalnych. Na przykład jedną z takich finalnych klas jest `String`. Dlaczego? Cóż... wyobraź sobie chaos, jaki mógłby powstać, gdyby ktoś zmienił działanie tej klasy!

P: Czy można oznaczyć metodę jako finalną bez jednoczesnego oznaczania całej klasy jako finalnej?

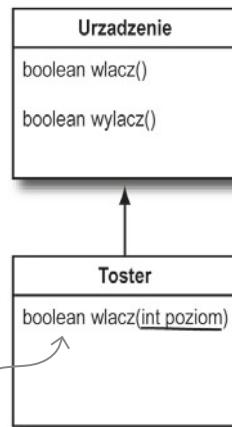
O: Jeśli chcesz uniemożliwić przesłonięcie konkretnej *metody*, oznacz ją jako finalną (używając w tym celu modyfikatora `final`). Jeśli natomiast chcesz mieć pewność, że żadna z metod nie zostanie przesłonięta, oznacz całą *klasę* jako finalną.

Jak dotrzymać kontraktu — reguły przesłaniania

Przesłaniając metodę klasy bazowej, wyrażasz zgodę na dotrzymanie warunków kontraktu. Kontraktu, który przykładowo stwierdza: „nie pobieram żadnych argumentów i zwracam wartość logiczną”. Innymi słowy, zarówno argumenty, jak i typ wartości wynikowej tworzonej metodą muszą być dokładnie takie same jak w przesłanianej metodzie klasy bazowej.

Kontraktem są metody.

Jeśli polimorfizm ma działać, to wersja przesłoniętej metody klasy `Urzadzenie` zdefiniowana w klasie `Toster` musi działać podczas wykonywania programu. Pamiętaj, że kompilator sprawdza typ odwołania, aby upewnić się, że posługując się danym odwołaniem, możesz wywołać konkretną metodę. W przypadku odwołania typu `Urzadzenie` do obiektu klasy `Toster` kompilator zwraca uwagę wyłącznie na to, czy klasa `Urzadzenie` dysponuje metodą, którą starasz się wywołać, używając do tego odwołania typu `Urzadzenie`. Jednak w trakcie działania programu wirtualna maszyna Javy nie sprawdza typu *odwołania* (`Urzadzenie`), lecz faktyczny obiekt `Toster` przechowywany na stercie. A zatem, jeśli kompilator już *stwierdził poprawność* wywołania metody, to może ona działać wyłącznie w sytuacji, gdy metoda przesłaniająca będzie mieć takie same typy argumentów i wartości wynikowej. W przeciwnym razie, posiadając odwołanie do obiektu `Urzadzenie`, ktoś mógłby wywołać bezargumentową wersję metody `wlacz()`, nawet jeśli w klasie `Toster` metoda o tej nazwie będzie wymagać przekazania liczby całkowitej. Która z nich zostanie wywołana podczas działania programu? Metoda zdefiniowana w klasie `Urzadzenie`. Innymi słowy, ***metoda wlacz(int poziom) zdefiniowana w klasie Toster nie przesłoniła metody wlacz() bazowej klasy Urzadzenie!***



To nie jest przesłonięcie metody!

W metodzie przestanowiącej nie można zmieniać typów argumentów!

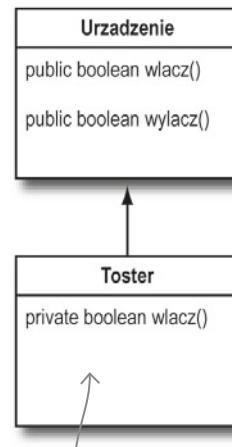
To jest dopuszczalne PRZECIĄŻENIE metody, ale nie jej przesłonięcie!

1 Typy argumentów i wartości wynikowej muszą być zgodne.

Kontrakt klasy bazowej określa, w jaki sposób inny kod może używać jej metod. Jeśli w klasie bazowej pewna metoda wymaga podania argumentu, to w klasie potomnej przesłaniającej tej metodę trzeba użyć takiego samego argumentu. A jeśli w klasie bazowej metoda zwraca wartość pewnego typu, to ta sama metoda przesłaniana w klasie potomnej musi zwracać wartość tego samego typu lub typu potomnego. Pamiętaj, że obiekt klasy potomnej na pewno będzie w stanie wykonywać wszystkie operacje zadeklarowane w klasie bazowej; dlatego też zwracanie klasy potomnej tam, gdzie oczekiwany jest obiekt jej klasy bazowej, nie niesie ze sobą żadnego niebezpieczeństwa.

2 Nie można ograniczyć dostępu do metody.

Oznacza to, że poziom dostępu do metody musi być taki sam lub większy (mniej restrykcyjny). Wynika z tego, że nie możesz, na przykład, przesłonić metody publicznej i zdefiniować jej jako prywatnej. Jaki szok musiałby przeżyć ktoś, kto wywołuje metodę, która jest publiczna (podczas komplikacji), gdyby nagle okazało się, że podczas działania programu wirtualna maszyna Javy zamknie mu drzwi przed nosem, gdyż nowa wersja metody, przesłaniająca oryginalną, jest prywatna!



TO NIE JEST DOPUSZCZALNE!
Taki sposób przestanowienia metody nie jest dopuszczalny, gdyż zmieniasz poziom dostępu. Jednocześnie nie jest to także dopuszczalny sposób przeciążania metody, gdyż nie zostaty zmienione typy jej argumentów.

Przeciążanie metody

Przeciążanie metod to nic innego jak tworzenie metod o tej samej nazwie, lecz różnych listach argumentów. I tyle. Przeciążanie metod nie ma nic wspólnego z polimorfizmem!

Przeciążanie pozwala na tworzenie różnych wersji tej samej metody, które ze względu na wygodę używającego ich programisty różnią się listami argumentów. Na przykład, jeśli dysponujesz metodą, która wymaga podania jedynie liczby całkowitej, to, przykładowo, przed jej wywołaniem programista, który chce jej użyć, musi skonwertować argument będący liczbą zmiennoprzecinkową do postaci liczby całkowitej. Jeśli jednak przeciążysz tę metodę i stworzysz jej wersję akceptującą argument będący liczbą zmiennoprzecinkową, to ułatwisz życie osobie, która chce jej użyć. Ponownie zajmiemy się tym zagadnieniem przy okazji omawiania konstruktorów w rozdziale poświęconym cyklowi życia obiektów.

Ponieważ przeciążona metoda nie stara się spełnić „polimorficznego kontraktu” zdefiniowanego przez klasę bazową, zatem metody przeciążone są znacznie bardziej elastyczne.

1 Typy wartości wynikowych mogą być inne.

W metodach przeciążonych można bez przeszkód zmieniać typy wartości wynikowych, o ile tylko listy argumentów będą różne.

2 Nie możesz zmieniać jedynie typu wartości wynikowej.

Jeśli metody różnią się wyłącznie typem wartości wynikowej, to kompilator nie potraktuje tego jako przeciążania, lecz uzna, że starasz się *przesłonić* w klasie potomnej. Lecz nawet w tym przypadku może pojawić się błąd, jeśli typ wartości wynikowej nie będzie typem potomnym wartości wynikowej zadeklarowanej w klasie bazowej. Aby przeciążyć metodę, MUSISZ w niej zmienić listę argumentów, a jednocześnie możesz dowolnie zmienić jej wartość wynikową.

3 Możesz dowolnie zmieniać poziom dostępu.

Możesz tworzyć przeciążone wersje metod, do których dostęp jest bardziej ograniczony. Nie ma to żadnego znaczenia, gdyż nowa metoda nie musi spełniać kontraktu określonego przez metodę przeciążaną.

Metody przeciążone to zupełnie różne metody, które mają identyczne nazwy. Przeciążanie metod nie ma nic wspólnego z dziedziczeniem ani polimorfizmem. Przeciążanie metod nie jest także tym samym, co ich przesłanianie.

Przykłady poprawnego i dopuszczalnego przeciążania metod:

```
class Przeciazanie {

    String unikalnyID;

    public int dodaj(int a, int b) {
        return a + b;
    }

    public double dodaj(double a, double b) {
        return a + b;
    }

    public void setID(String id) {
        // bardzo dużo kodu sprawdzającego, a potem:
        unikalnyID = id;
    }

    public void setID(int numerSr) {
        String numS = "" + numerSr;
        setID(numS);
    }
}
```

Ćwiczenia. Pomieszane komunikaty



Ćwiczenia Pomieszane komunikaty

```
a = 6;    ↗56
b = 5;    ↗11
a = 5;    ↗65
```

Poniżej przedstawiony został krótki program napisany w Javie. Brakuje w nim jednego fragmentu. Twoim zadaniem jest dopasowanie jednego z proponowanych bloków kodu (przedstawionych z lewej strony) do wyników, które program wygenerowałby po wstawieniu wybranego bloku. Zauważ, że nie wszystkie wiersze wyników zostaną wykorzystane, a niektóre z nich mogą być wykorzystane więcej niż jeden raz. Narysuj linie łączące bloki kodu z odpowiadającymi im wynikami.

Program:

```
class A {
    int izm = 7;
    void m1() {
        System.out.print("A - m1, ");
    }

    void m2() {
        System.out.print("A - m2, ");
    }

    void m3() {
        System.out.print("A - m3, ");
    }
}

class B extends A {
    void m1() {
        System.out.print("B - m1, ");
    }
}
```

```
class C extends B {
    void m3() {
        System.out.print("C - m3, "+ (izm + 6));
    }
}
```

```
public class PomieszaneKomunikaty7 {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        C c = new C();
        A a2 = new C();
    }
}
```

tutaj należy wstawić proponowany fragment kodu (3 wiersze).

Proponowane fragmenty kodu:

```
b.m1();    }
c.m2();    }
a.m3();    }
```

```
c.m1();    }
c.m2();    }
c.m3();    }
```

```
a.m1();    }
b.m2();    }
c.m3();    }
```

```
a2.m1();   }
a2.m2();   }
a2.m3();   }
```

Wyniki:

A - m1, A - m2, C - m3, 6

B - m1, A - m2, A - m3,

A - m1, B - m2, A - m3,

B - m1, A - m2, C - m3, 13

B - m1, C - m2, A - m3,

B - m1, A - m2, C - m3, 6

A - m1, A - m2, C - m3, 13



Ćwiczenie

BĄDŹ kompilatorem



Która z par metod A i B przedstawionych po prawej stronie, po wstawieniu do klas przedstawionych po stronie lewej, dałaby się skompilować, a w przypadku uruchomienia wygenerowałaby wyniki zamieszczone u dołu strony? Metoda A ma zostać wstawiona do klasy Potwor, a metoda B do klasy Wampir.

```
public class PotworTester {
    public static void main(String[] args) {
        Potwor[] ptw = new Potwor[3];
        ptw[0] = new Wampir();
        ptw[1] = new Smok();
        ptw[2] = new Potwor();
        for (int x = 0; x < 3; x++) {
            ptw[x].strasz(x);
        }
    }
}

class Potwor {
    A
}

class Wampir extends Potwor {
    B
}

class Smok extends Potwor {
    boolean strasz(int stopien) {
        System.out.println("zioń ogniem");
        return true;
    }
}
```

```
Wiersz polecenia
T:\>java PotworTester
można gryza?
zioń ogniem
ałłuuuuu
```

- 1 boolean strasz(int d) {
 A System.out.println("ałłuuuuu");
 return true;
 }

 B System.out.println("można gryza?");
 return false;
 }
- 2

 A boolean strasz(int x) {
 System.out.println("ałłuuuuu");
 return true;
 }

 B int strasz(int f) {
 System.out.println("można gryza?");
 return 1;
 }
- 3

 A boolean strasz(int x) {
 System.out.println("ałłuuuuu");
 return false;
 }

 B boolean strasz(int x) {
 System.out.println("można gryza?");
 return true;
 }
- 4

 A boolean strasz(int z) {
 System.out.println("ałłuuuuu");
 return true;
 }

 B boolean strasz(byte b) {
 System.out.println("można gryza?");
 return true;
 }

Zagadka. Zagadkowy basen



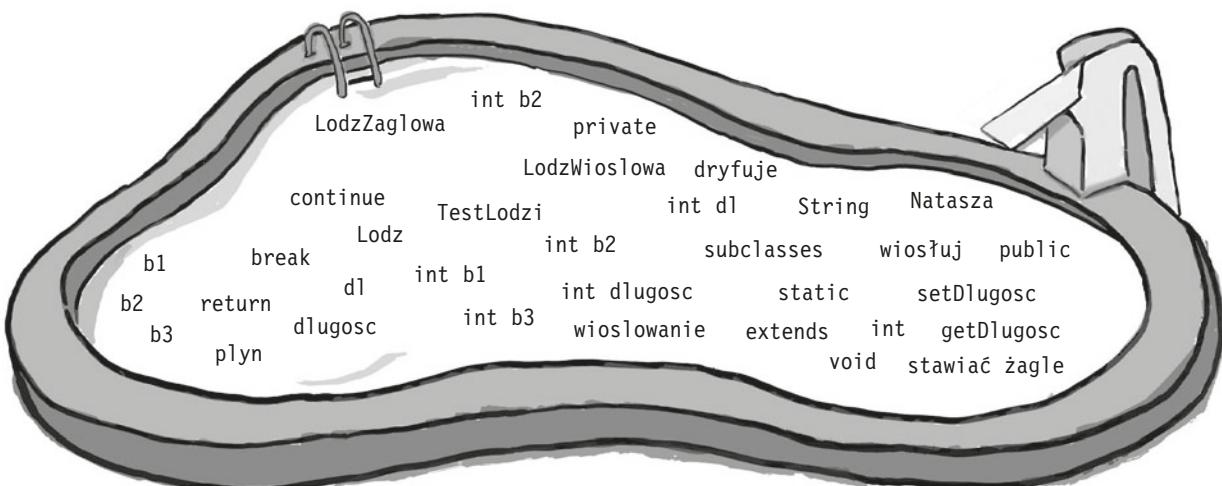
Zagadkowy basen

Twoim zadaniem jest wybranie fragmentów kodu z basenu i umieszczenie ich w miejscach kodu oznaczonych podkreśleniami. Ten sam fragment kodu może być użyty więcej niż jeden raz, jednak może się zdarzyć, że nie wszystkie fragmenty zostaną wykorzystane. Zadanie polega na stworzeniu klasy, którą będzie można skompilować i która wygeneruje wyniki przedstawione poniżej. Nie daj się zwieść pozorom — ta zagadka jest trudniejsza, niż można by przypuszczać.

```
public class LodzWioslowa _____ {  
    public _____ wioslowanie() {  
        System.out.println("wiosłuj Natasza");  
    }  
}  
  
public class _____ {  
    private int _____;  
    _____ void _____( _____ ) {  
        dlugosc = dl;  
    }  
    public int getDlugosc() {  
        _____ _____;  
    }  
    public _____ plyn() {  
        System.out.print("_____");  
    }  
}
```

```
public class TestLodzi {  
    _____ _____ main(String[] args) {  
        _____ b1 = new Lodz();  
        LodzZaglowa b2 = new _____();  
        LodzWioslowa _____ = new LodzWioslowa();  
        b2.setDlugosc(32);  
        b1._____();  
        b3._____();  
        _____ .plyn();  
    }  
}  
  
public class _____ Lodz {  
    public _____ _____() {  
        System.out.print("_____");  
    }  
}
```

Wyniki: dryfuje dryfuye stawiać żagle

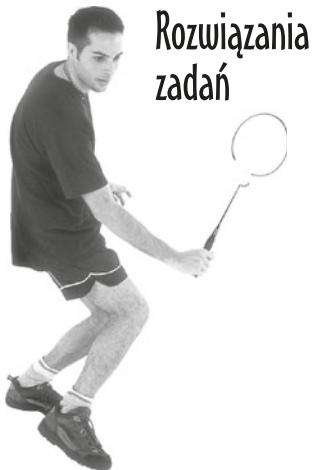




BĄDŹ kompilatorem

Para 1. Będzie działać.

Para 2. Nie da się skompilować ze względu na to, że metoda strasz() klasy Wampir zwraca wartość wynikową (typu int).



Metoda strasz() (B) zdefiniowana w klasie Wampir nie jest ani właściwym przeciążeniem, ani właściwym przesłonięciem analogicznej metody z klasy Potwor. Zmiana WYŁĄCZNIE typu zwracanej wartości wynikowej nie wystarcza, by prawidłowo przeciążyć metodę; a ponieważ typ int nie jest zgodny z typem boolean, zatem także przesłonięcie nie będzie prawidłowe. (Pamiętaj, że jeśli zmieniasz JEDYNIE typ wartości wynikowej, to musi on być zgodny z typem wynikowym metody zdefiniowanej w klasie bazowej; dopiero w takim przypadku przesłonięcie będzie prawidłowe.)

Para 3. i 4. Można je skompilować, jednak zwracane wyniki będą mieć postać:

ałłuuuuu

zioń ogniem

ałłuuuuu

Pamiętaj, że w klasie Wampir nie została przesłonięta metoda strasz() klasy Potwor. (Metoda strasz() klasy Wampir przedstawiona w 4. parze pobiera argument typu byte, a nie typu int).

Propozowane fragmenty kodu:

Ćwiczenia
Pomieszczone komunikaty

```
b.m1();
c.m2();
a.m3(); }
```

```
c.m1();
c.m2();
c.m3(); }
```

```
a.m1();
b.m2();
c.m3(); }
```

```
a2.m1();
a2.m2();
a2.m3(); }
```

Wyniki:

A – m1, A – m2, C – m3, 6

B – m1, A – m2, A – m3,

A – m1, B – m2, A – m3,

B – m1, A – m2, C – m3, 13

B – m1, C – m2, A – m3,

B – m1, A – m2, C – m3, 6

A – m1, A – m2, C – m3, 13



```
public class LodzWioslowa extends Lodz {
    public void wioslowanie() {
        System.out.println("wiosłuj Natasza");
    }
}

public class Lodz {
    private int dlugosc;
    public void setDlugosc ( int dl ) {
        dlugosc = dl;
    }
    public int getDlugosc() {
        return dlugosc;
    }
    public void plyn() {
        System.out.print("dryfuje");
    }
}

public class TestLodzi {
    public static void main(String[] args) {
        Lodz b1 = new Lodz();
        LodzZaglowa b2 = new LodzZaglowa();
        LodzWioslowa b3 = new LodzWioslowa();
        b2.setDlugosc(32);
        b1.plyn();
        b3.plyn();
        b2.plyn();
    }
}

public class LodzZaglowa extends Lodz {
    public void plyn() {
        System.out.print("stawiać żagle");
    }
}
```

Wyniki: dryfuje dryfuje stawiać żagle

8. Interfejsy i klasy abstrakcyjne

Poważny polimorfizm

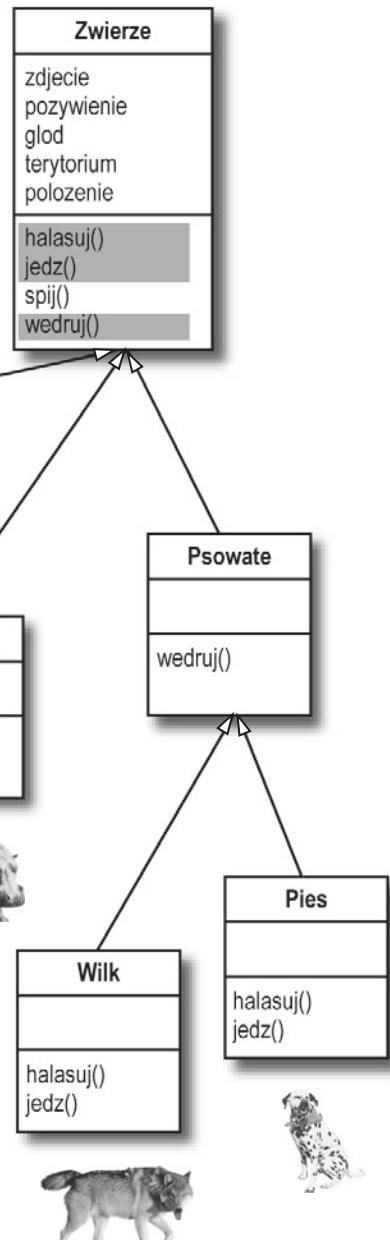


Dziedziczenie to jedynie początek. Aby w pełni wykorzystać możliwości, jakie daje polimorfizm, będziemy potrzebować interfejsów (i to bynajmniej nie graficznych interfejsów użytkownika). Musimy zostawić proste dziedziczenie i pójść dalej — dotrzeć do poziomu elastyczności i możliwości rozbudowy kodu, jakie daje jedynie projektowanie i programowanie przy wykorzystaniu specyfikacji interfejsów. Niektóre z najfajniejszych możliwości Javy w ogóle nie byłyby dostępne, gdyby nie istniały interfejsy. Dlatego, nawet jeśli sam nie projektujesz klas, korzystając z interfejsów, to i tak musisz ich używać. Niemniej jednak będzie Ci zależeło, aby korzystać z interfejsów. Będziesz ich potrzebować przy projektowaniu swoich klas i **będziesz się zastanawiać, jak wcześniej mogłeś bez nich żyć.** Jednak co to jest interfejs? Otóż interfejs to w całkowicie — w 100% — abstrakcyjna klasa. A co to jest abstrakcyjna klasa? To klasa, która nie daje możliwości tworzenia obiektów. A do czego taka klasa może nam się przydać? O tym dowiesz się już niedługo. Jeśli jednak przypomnisz sobie informacje podane pod koniec poprzedniego rozdziału i zastanowisz się, w jaki sposób wykorzystaliśmy argumenty polimorficzne, aby jedna klasa *Weterynarz* mogła akceptować argumenty dowolnych typów rozszerzających klasę *Zwierze*, to... cóż, to był jedynie wierzchołek góry lodowej. Interfejsy stanowią **poli** w polimorfizmie, **ab** w abstrakcji, i **kofeinę** w Javie¹.

¹ W języku angielskim słowo „java” jest czasami używane jako ogólne określenie kawy, zwłaszcza kawy bardzo wysokiej jakości. — *przyp. tłum.*

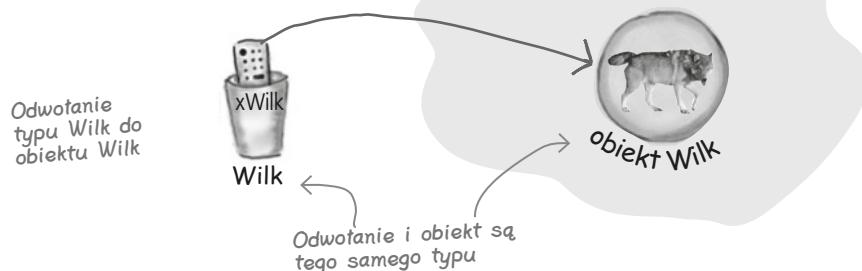
Czy projektując przedstawioną obok hierarchię klas, o czym zapomnieliśmy?

Struktura klas nie jest zła. Zaprojektowaliśmy ją w taki sposób, że powtarzanie się kodu zostało ograniczone do minimum i przesłoniliśmy metody, które według nas powinny być zaimplementowane w klasach potomnych. Z punktu widzenia polimorfizmu projekt jest dobry i elastyczny, można bowiem zaprojektować program w taki sposób, aby stosować w nim argumenty typu *Zwierze* (i deklaracje tablic); dzięki temu w trakcie działania programu będzie można użyć i przekazać do niego dowolne klasy potomne klasy *Zwierze* — *i to nawet takie, o których w ogóle nam się nie śniło podczas tworzenia kodu*. Stworzyliśmy też wspólny protokół dla wszystkich zwierząt (cztery metody, które chcemy, aby były znane całemu światu i były dostępne we wszystkich klasach zwierząt); został on zdefiniowany w klasie bazowej *Zwierze*. Teraz jesteśmy gotowi, by tworzyć nowe lwy, tygrysy, hipopotamy... i inne obiekty. O rany!

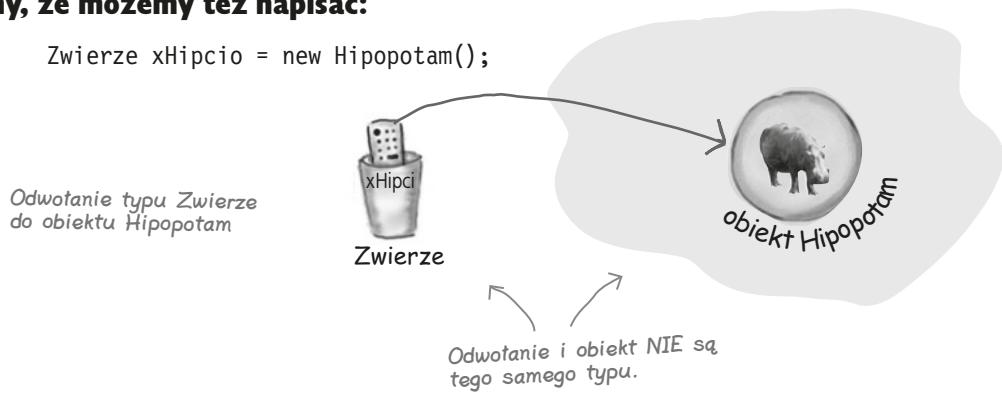


Wiemy, że możemy napisać:

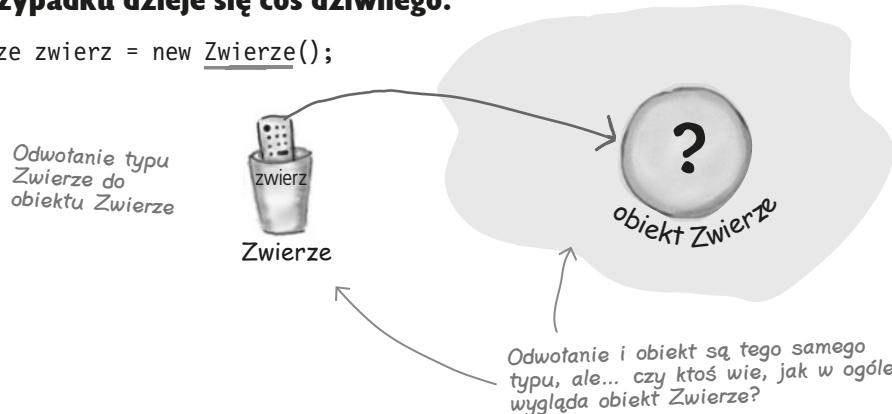
```
Wilk xWilk = new Wilk();
```

**Wiemy, że możemy też napisać:**

```
Zwierze xHipcio = new Hipopotam();
```

**Ale w tym przypadku dzieje się coś dziwnego:**

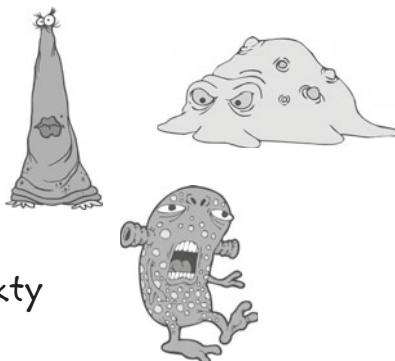
```
Zwierze zwierz = new Zwierze();
```



Kiedy obiekty schodzą na złą drogę

Jak wygląda obiekt utworzony przy użyciu new Zwierze()?

przerażające obiekty



Jakie są wartości jego składowych?

Obiektów niektórych klas po prostu nie należy tworzyć.

Utworzenie obiektu klasy Wilk, Hipopotam lub Tygrys ma sens, ale czym właściwie *jest* obiekt Zwierze? Jak wygląda? Jaki ma kolor, wielkość, ile ma nóg...?

Próba utworzenia obiektu Zwierze może przypominać **koszmarne skutki błędu transportera z serialu StarTrek™**, na przykład, gdyby podczas trwania procesu coś złego stało się z buforem.

Ale jak mamy rozwiązać ten problem? Klasa Zwierze jest nam *potrzebna*, gdyż korzystają z niej mechanizmy dziedziczenia i polimorfizmu. Chcemy jednak, aby programiści tworzyli jedynie obiekty mniej abstrakcyjnych *klas potomnych*, a nie obiekty samej klasy Zwierze. Chcemy, aby można było tworzyć obiekty Tygrys i Lew, *ale nie Zwierze*.

Na szczęście istnieje prosty sposób, który pozwala uniemożliwić tworzenie obiektów danej klasy; innymi słowy, sposób, który zabrania stosowania operatora **new** wraz z danym typem. Otóż, jeśli oznaczymy klasę modyfikatorem **abstract**, kompilator nie pozwoli, by jakikolwiek kod, w jakimkolwiek miejscu programu kiedykolwiek stworzył obiekt danej klasy.

Takiego typu abstrakcyjnego wciąż jednak można używać do deklarowania zmiennych referencyjnych. W rzeczywistości to jeden z głównych powodów, dla którego w ogóle istnieją klasy abstrakcyjne (możliwość stosowania ich jako argumentów polimorficznych, wartości wynikowych lub tablic polimorficznych).

Projektując hierarchię klas, musisz zdecydować, które klasy są *abstrakcyjne*, a które *konkretnie*. Klasy konkretne to te, które są na tyle szczegółowe, aby tworzyć ich obiekty. Fakt, że klasa jest *konkretna*, oznacza, że tworzenie jej obiektów ma sens.

Stworzenie klasy abstrakcyjnej jest łatwe — wystarczy umieścić przed deklaracją klasy słowo kluczowe **abstract**:

```
abstract class Psowate extends Zwierze {  
    public void wedruj() { }  
}
```

Kompilator nie pozwoli utworzyć obiektów klasy abstrakcyjnej

Fakt, iż klasa jest abstrakcyjna, oznacza, że nikt i nigdy nie będzie mógł utworzyć obiektu tej klasy. Wciąż można używać tej klasy jako typu deklarowanych zmiennych referencyjnych oraz w zastosowaniach związanych z polimorfizmem, jednak nie trzeba się martwić tym, że ktoś utworzy obiekt tej klasy. To gwarantuje nam kompilator.

```
abstract public class Psowate extends Zwierze
{
    public void wedruj() { }

}

public class UtworzPsowate {
    public void doDziela() {
        Psowate p;
        p = new Pies();
        p = new Psowate();
        p.wedruj()
    }
}
```

Ten fragment jest w porządku. Zawsze można przypisać obiekt klasy potomnej do zmiennej klasy bazowej, nawet jeśli klasa bazowa jest abstrakcyjna.

Klasa Psowate jest klasą abstrakcyjną, zatem kompilator nie pozwoli wykonać tej instrukcji.

```
Wiersz polecenia
T:\>javac UtworzPsowate.java
UtworzPsowate.java:17: Psowate is abstract; cannot be instantiated
    p = new Psowate();
1 error
```

Klasa abstrakcyjna nie ma właściwie* żadnego zastosowania, żadnej wartości ani żadnego sensu istnienia za wyjątkiem jej **rozszerszania**.

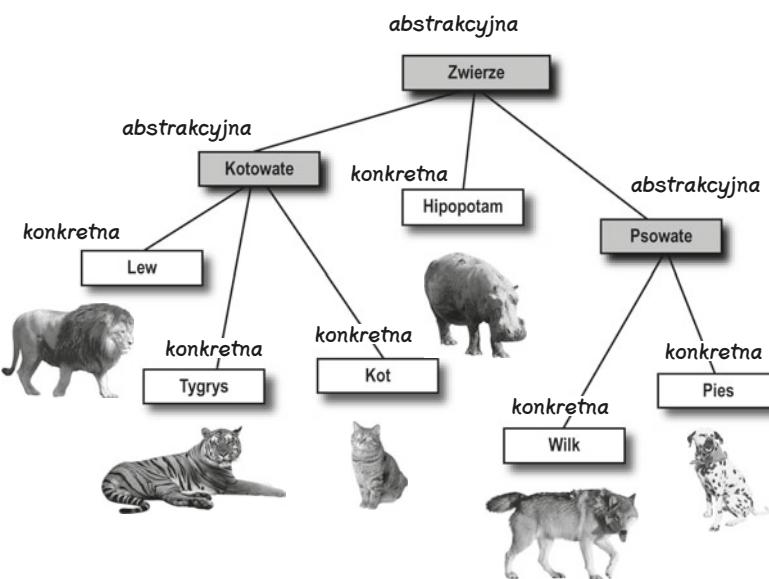
W przypadku klas abstrakcyjnych „goście”, którzy wykonują całą robotę podczas działania programu, to **obiekty klas potomnych** klasy abstrakcyjnej.

*) Jest jeden wyjątek od tej zasady — klasa abstrakcyjna może mieć składowe statyczne (więcej informacji na ten temat znajdziesz w rozdziale 10.).

Abstrakcyjne kontra konkretne

Klasa, która nie jest abstrakcyjna, nosi nazwę klasy konkretnej. W drzewie dziedziczenia zwierząt, jeśli klasy Zwierze, Psowate i Kotowane zdefiniujemy jako klasy abstrakcyjne, to konkretnymi będą klasy stanowiące „liście” tego drzewa — Hipopotam, Wilk, Pies, Lew, Tygrys oraz Kot.

Przejrzyj Java API, a znajdziesz wiele klas abstrakcyjnych; zwłaszcza w bibliotece związanej z tworzeniem graficznego interfejsu użytkownika. Jak wygląda klasa Component? Jest to klasa bazowa wielu klas reprezentujących różne elementy graficznego interfejsu użytkownika, takich jak na przykład: przyciski, wielowierszowe pola tekstowe, paski przewijania, okna dialogowe. Nie tworzy się jednak obiektu *ogólnej* klasy Component i nie wyświetla się go na ekranie, można to natomiast zrobić z obiektem klasy JButton. Innymi słowy, tworzone są jedynie obiekty *konkretnych klas potomnych* klasy Component, nigdy natomiast samej klasy Component.



WYSIL SZARE KOMÓRKI



Abstrakcyjna czy konkretna?

Skąd wiadomo, czy dana klasa powinna być abstrakcyjna? Klasa **Wino** prawdopodobnie powinna być abstrakcyjna. Ale co z klasami **WinoCzerwone** i **WinoBiale**? Zapewne także i one powinny być klasami abstrakcyjnymi (przynajmniej dla niektórych spośród nas). Jednak w jakim miejscu hierarchii powinny się zaczynać klasy konkretne?

Czy **PinotNoir** ma być klasą konkretną czy abstrakcyjną? Można sądzić, że Pinot Noir Camelot rocznik 1997 na pewno będzie klasą konkretną. Ale skąd możemy mieć taką pewność?

Spójrzmy na przedstawione powyżej drzewo dziedziczenia klas reprezentujących zwierzęta. Czy dokonany przez nas wybór klas abstrakcyjnych i konkretnych wydaje się właściwy? Czy chciałbyś cokolwiek zmienić w tej strukturze (oczywiście za wyjątkiem dodania nowych zwierząt)?

Metody abstrakcyjne

Oprócz klas można także tworzyć abstrakcyjne *metody*. Stworzenie klasy abstrakcyjnej oznacza, że trzeba będzie ją *rozszerzyć*, z kolei utworzenie metody abstrakcyjnej informuje o konieczności jej *przesłonięcia*. Mógłbyś dojść do wniosku, że niektóre (lub wszystkie) zachowania klasy abstrakcyjnej powinny być zaimplementowane w bardziej szczegółowej klasie potomnej. Innymi słowy oznaczałoby to, że nie możesz wyobrazić sobie implementacji żadnej z ogólnych metod klasy bazowej, które mogłyby być wykorzystane w klasach potomnych. Jak mogłyby wyglądać taka ogólna metoda `jedz()`?

Metoda abstrakcyjna nie ma treści!

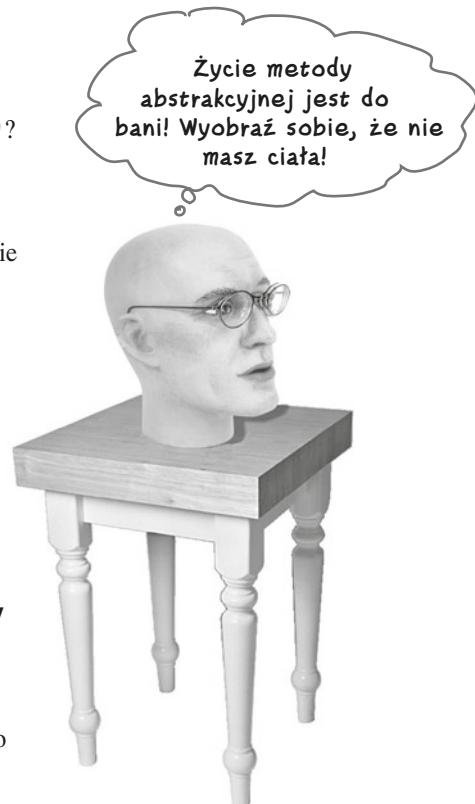
Ponieważ już zdecydowałeś, że nie ma takiego kodu, którego umieszczenie w metodzie abstrakcyjnej miałoby sens, zatem nie umieścisz w niej żadnej treści. Z tego względu w deklaracji metody nie powinieneś zapisywać nawiasów klamrowych — wystarczy, że zakończysz ją średnikiem.

```
public abstract void jedz();
```

Nie ma treści metody!
Zakończ deklarację średnikiem.

Jeśli zadeklarujesz metodę jako abstrakcyjną, w ten sam sposób MUSISZ zadeklarować całą klasę. Abstrakcyjne metody nie mogą istnieć w „nieabstrakcyjnych” klasach.

Jeśli w klasie znajdzie się chociaż jedna metoda abstrakcyjna, to klasę trzeba oznaczyć jako abstrakcyjną. Jednak w klasie abstrakcyjnej *można* umieszczać zarówno abstrakcyjne, jak i normalne metody.



Nie istnieją głupie pytania

P: Do czego są potrzebne metody abstrakcyjne?
Uważalem, że klasy abstrakcyjne istnieją tylko po to, aby można w nich było umieszczać wspólny kod, który będą mogły dziedziczyć klasy potomne.

O: Implementacje metod (czyli metody mające zawartość) doskonale nadają się do umieszczania w klasach bazowych. Jeśli takie rozwiązanie ma sens. Jednak w przypadku klas abstrakcyjnych często zdarza się, że rozwiązanie to *nie jest zasadne*, gdyż nie można stworzyć ogólnego kodu, który byłby użyteczny we wszystkich klasach potomnych. Metody abstrakcyjne tworzone są w innym celu. Otóż, choć nie umieszcza się w nich żadnego konkretnego kodu, to jednak wciąż pozwalają one na zdefiniowanie protokołu dla pewnej grupy podtypów (podklaśs).

P: Co jest przydatne, gdy...

O: Oczywiście ze względu na polimorfizm! Pamiętaj, że naszym celem jest uzyskanie możliwości wykorzystania typu klasy bazowej (często klasy abstrakcyjnej) jako typu argumentów, wyników zwracanych przez metody oraz typu tablic. Dzięki temu możemy dodawać do programu nowe podtypy (na przykład klasy potomne klasy *Zwierze*) bez konieczności dodawania metod przeznaczonych do obsługi tych typów. Wyobraź sobie, jakie zmiany musiałbyś wprowadzać w kodzie klasy *Weterynarz*, gdyby jej metody nie akceptowały argumentu typu *Zwierze*. Musiałbyś tworzyć osobną metodę dla każdej klasy potomnej klasy *Zwierze*! Osobną dla klasy *Lew*, osobną dla *Tygrys* i tak dalej... Już rozumiesz? A zatem, zadeklarowanie metody abstrakcyjnej sprowadza się do stwierdzenia: „*Wszystkie podtypy tego typu mają TĘ metodę*”, co ma jedynie dobre skutki dla polimorfizmu.

Metody abstrakcyjne należy implementować

MUSISZ zaimplementować wszystkie metody abstrakcyjne



Implementacja metod abstrakcyjnych jest bardzo podobna do przesłaniania metod.

Metody abstrakcyjne nie mają treści, istnieją wyłącznie po to, aby stosować je w rozwiązańach związanych z polimorfizmem. To oznacza, że pierwsza konkretna klasa w drzewie dziedziczenia musi implementować *wszystkie* metody abstrakcyjne.

Zadanie to można jednak przerzucić na kogoś innego, tworząc kolejną klasę abstrakcyjną. Jeśli zarówno klasa *Zwierze*, jak i *Psowate* są klasami abstrakcyjnymi, to w klasie *Psowate* nie trzeba implementować abstrakcyjnych metod klasy *Zwierze*. Jednak tworząc pierwszą konkretną klasę potomną, taką jak *Pies*, trzeba będzie zaimplementować w niej wszystkie metody abstrakcyjne klas *Zwierze* i *Psowate*.

Pamiętaj jednak, że klasy abstrakcyjne mogą mieć zarówno metody abstrakcyjne, jak i normalne — nieabstrakcyjne. Zatem klasa *Psowate* mógłaby implementować abstrakcyjne metody klasy *Zwierze*, dzięki czemu nie trzeba by było implementować ich w klasie *Pies*. Jednak gdyby klasa *Psowate* nie „zrobiła” niczego z abstrakcyjnymi metodami klasy *Zwierze*, to musiałyby one zostać zaimplementowane w klasie *Pies*.

Stwierdzając: „musisz zaimplementować metodę abstrakcyjną”, mamy na myśli *konieczność podania jej treści*. Oznacza to, że w klasie konkretnej musisz stworzyć metodę o tej samej sygnaturze (nazwie i argumentach) oraz typie wartości wynikowej zgodnym z typem wartości wynikowej zadeklarowanym w metodzie abstrakcyjnej. Zawartość metody zależy oczywiście od Ciebie. Java zwraca uwagę wyłącznie na to, aby odpowiednia metoda *istniała*.



Zaostrz ołówek

Klasy abstrakcyjne kontra klasy konkretne

Znajdźmy jakieś praktyczne zastosowanie dla tej całej abstrakcyjnej gadańiny. Poniżej, w środkowej kolumnie podane zostały nazwy klas. Twoim zadaniem jest wymyślenie zastosowań, w których dana klasa mogłaby być konkretna, oraz takich, w których mogłaby być klasą abstrakcyjną. Aby łatwiej Ci było zacząć, pierwszych kilka przykładów wypełniliśmy sami. Na przykład w programie pielęgnacji roślin klasa Drzewo byłaby klasą abstrakcyjną, gdyż różnice pomiędzy poszczególnymi klasami, na przykład Sosna i Brzoza, mają znaczenie. Jednak w programie do symulacji gry w golfa, Drzewo mogłaby być klasą konkretną (zapewne rozszerzającą klasę Przeszkoda), gdyż program nie musi rozróżniać poszczególnych rodzajów drzew. (To ćwiczenie nie ma jednego, konkretnego rozwiązania — zależy ono od rozwiązania, jakie przyjmiesz).

Klasa konkretna

symulacja gry w golfa

program przetwarzania zdjęć satelitarnych

Koszykarz

Krzesło

Klient

ZamowienieSprzedazy

Ksiazka

Sklep

Dostawca

PoleGolfowe

Gaznik

Piekarnik

Klasa

Drzewo

Dom

Miasto

Koszykarz

Krzesło

Klient

ZamowienieSprzedazy

Ksiazka

Sklep

Dostawca

PoleGolfowe

Gaznik

Piekarnik

Klasa abstrakcyjna

program pielęgnacji roślin

program projektowy

aplikacja trenerska

Polimorfizm w działaniu

Załóżmy, że chcemy stworzyć własną klasę listy — listę przechowującą obiekty Pies (udajmy przy tym na chwilę, że nie znamy klasy ArrayList). Na początku stworzymy w niej wyłącznie metodę dodaj(). Do przechowywania obiektów Pies wykorzystamy zwyczajną tablicę (Pies[]) o wielkości 5 elementów. Kiedy limit pięciu obiektów zostanie wyczerpany, metodę dodaj() będzie można wywoływać, lecz nie będzie ona wykonywać żadnych operacji. W przypadku, gdy limit obiektów nie został wyczerpany, metoda doda obiekt w pierwszej wolnej komórce tablicy, po czym powiększy o jeden indeks dostępnej komórki (nastepnyIndx).

Stwórz własną listę do przechowywania obiektów Pies

(Zapewne najgorszy z możliwych przykładów samodzielnego tworzenia klasy przypominającej ArrayList).

```
public class MojaListaPsow {  
    private Pies[] psy = new Pies[5]; ← W niewidoczny sposób wykorzystujemy zwyczajną tablicę typu Pies.  
    private int nastepnyIndx = 0; ← Ta składowa będzie inkrementowana za każdym razem, gdy do tablicy zostanie dodany nowy obiekt.  
  
    MojaListaPsow  
    {  
        Pies[] psy  
        int nastepnyIndx  
    }  
  
    public void dodaj(Pies p) {  
        if (nastepnyIndx < psy.length) { } ← Jeżeli cała tablica nie została wypełniona wcześniej, dodajemy do niej nowy obiekt Pies i wyświetlamy komunikat.  
        psy[nastepnyIndx] = p;  
        System.out.println("Pies dodany na pozycji nr " + nastepnyIndx);  
        nastepnyIndx++; ← Inkrementujemy, aby znać wartość następnego indeksu tablicy.  
    }  
}
```

O rany... teraz musimy przechowywać także obiekty Kot!

W tym przypadku mamy kilka różnych możliwości:

- 1) Stworzyć osobną listę — `MojaListaKotow` — przechowującą obiekty klasy `Kot`. Rozwiązanie niezbyt eleganckie.
- 2) Stworzyć jedną klasę — `MojaListaPsowIKotow` — która będzie zawierać dwie niezależne tablice i udostępniać dwie różne metody `dodaj()` — `dodajPsa(Pies p)` oraz `dodajKota(Kot k)`.
- 3) Stworzyć ogólną klasę `ListaZwierząt`, w której będzie można przechowywać obiekty dowolnych klas potomnych klasy `Zwierze` (ponieważ doskonale wiemy, że jeśli do specyfikacji programu zostały dodane obiekty klasy `Kot`, to wcześniej czy później zostaną dodane także obiekty *innych* klas). To rozwiązanie najbardziej nam się podoba, zatem zmodyfikujmy naszą klasę w taki sposób, aby była bardziej ogólna, a konkretnie, aby pozwalała na przechowywanie obiektów klasy `Zwierze`, a nie `Pies`. W przedstawionym poniżej kodzie wprowadzone modyfikacje zostały wyróżnione ciemniejszym tłem (sama logika działania klasy nie uległa zmianie, w całym kodzie zmieniliśmy jedynie typy z `Pies` na `Zwierze`).

Stwórz własną listę do przechowywania obiektów Zwierze

```

public class MojaListaZwierząt {
    private Zwierze[] zwierzeta = new Zwierze[5];
    private int następnyIndx = 0;

    public void dodaj(Zwierze z) {
        if (następnyIndx < zwierzeta.length) {
            zwierzeta[następnyIndx] = z;
            System.out.println("Zwierze dodano na pozycji nr " + następnyIndx);
            następnyIndx++;
        }
    }
}

public class MojaListaZwierzątTester {
    public static void main(String[] args) {
        MojaListaZwierząt lista = new MojaListaZwierząt();
        Pies p = new Pies();
        Kot k = new Kot();
        lista.dodaj(p);
        lista.dodaj(k);
    }
}

```

Bez paniki! Nie tworzymy nowego obiektu `Zwierze`, a jedynie nową tablicę, typu `Zwierze`. (Pamiętaj, że nie możesz utworzyć obiektu klasy abstrakcyjnej, MOŻESZ natomiast utworzyć tablicę, która będzie ZAWIERAĆ obiekty tego typu).

A co z obiektami innych klas?

Dlaczego nie stworzyć listy, w której można by zapisać cokolwiek?

Doskonale wiesz, dokąd zmierzamy. Chcemy zmienić typ tablicy oraz argument metody `dodaj()` na jakąś klasę, która w hierarchii dziedziczenia jest gdzieś powyżej klasy `Zwierze`. Na klasę, która jest jeszcze *bardziej ogólna, bardziej abstrakcyjna*. Ale jak to zrobić? Przecież nie dysponujemy klasą bazową klasy `Zwierze`.

wersja 3

(To jedynie kilka metod udostępnianych przez klasę `ArrayList`... jest ich znacznie więcej).

No to jeszcze raz, może...

Czy przypominasz sobie metody klasy `ArrayList`? Zwróć uwagę, że wszystkie metody — `add()`, `contains()`, `indexOf()` — akceptują obiekty klasy... `Object`!

Każda klasa w Javie jest klasą potomną klasy `Object`.

Klasa `Object` jest „matką” — klasą bazową — *wszystkich* innych klas.

Nawet w przypadku wykorzystania polimorfizmu musiałbyś stworzyć klasę z metodami pobierającymi i zwracającymi *Twój* typ bazowy. Gdyby nie istniała wspólna klasa bazowa, twórcy Javy nie mieliby możliwości stworzenia klas posiadających metody akceptujące *Twoje* klasy... klas, o których nie mieli pojęcia, projektując i implementując klasę `ArrayList`.

A zatem od samego początku tworzyłeś klasy potomne klasy `Object` i nawet o tym nie wiedziałeś. **Każda klasa, którą stworzyłeś, rozszerza klasę `Object`**, choć nigdy nie musiałeś tego jawnie wyrażać. Jednak możesz to sobie wyobrazić w taki sposób, jak gdyby każda tworzona klasa wyglądała jak w poniższym przykładzie:

```
public class Pies extends Object { }
```

Ale zaraz, przecież klasa `Pies` już rozszerza inną klasę — `Psowate`. Nic nie szkodzi. W takim przypadku kompilator sprawi, że to klasa `Psowate` będzie rozszerzać klasę `Object`. Ale `Psowate` rozszerza klasę `Zwierze`. Nie ma sprawy. Zatem kompilator zapewni, że to klasa `Zwierze` będzie rozszerzać klasę `Object`.

Każda klasa, która jawnie nie rozszerza innej klasy, w niejawnym sposobie rozszerza klasę `Object`.

A ponieważ klasa `Pies` rozszerza klasę `Psowate`, zatem nie rozszerza bezpośrednio klasę `Object` (choć robi to pośrednio). To samo dotyczy klasy `Psowate`; natomiast klasa `Zwierze` bezpośrednio rozszerza klasę `Object`.

ArrayList

boolean remove(Object elem)
 Usuwa z listy obiekt przekazany jako parametr. Zwraca wartość true, jeśli obiekt znajdował się na liście.

boolean contains(Object elem)
 Zwraca true, jeśli obiekt przekazany jako parametr znajduje się na liście.

boolean isEmpty()
 Zwraca true, jeśli lista jest pusta.

int indexOf(Object elem)
 Zwraca indeks obiektu przekazanego jako parametr lub wartość -1.

boolean add(Object elem)
 Dodaje obiekt do listy (zwraca wartość true).

// i więcej

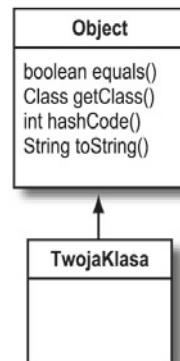
Wiele metod klasy `ArrayList` stosuje ostateczny typ polimorficzny — `Object`. Ponieważ każda klasa w Javie jest klasą potomną klasy `Object`, zatem w wywołaniach metod klasy `ArrayList` można przekazywać dowolne obiekty!

(Uwaga: w wersji 5.0 Javy metody `add()` i `get()` wyglądają w rzeczywistości nieco inaczej niż te przedstawione powyżej; jak na razie jednak możesz je sobie wyobrażać właściwie w taki sposób. Zajmiemy się tym nieco później).

Czym jest „superultramegaklasa” Object?

Wyobraź sobie, że jesteś Java, i powiedz, jakie są czynności, które byś chciał, aby *każdy* obiekt mógł wykonywać? Hmm... pomyślmy... może metoda, która pozwalałaby stwierdzić, czy jeden obiekt jest równy innemu? A co z metodą, która zwracałaby informacje o faktycznej klasie obiektu? Może metoda, która zwracałaby kod mieszący obiektu, tak aby można go było zapisywać w tablicy asocjacyjnej (do tego zagadnienia wróćmy jeszcze w rozdziale 16.). Oto jeszcze jedna dobra metoda — metoda pozwalająca na wyświetlenie komunikatu tekstowego o obiekcie.

Wiesz co? Okazuje się, że w jakiś magiczny sposób klasa Object faktycznie udostępnia metody dające wymienione wcześniej możliwości. Choć nie są to jedyne możliwości klasy Object, to właśnie one najbardziej nas interesują.



Niektóre metody klasy Object

Każda tworzona klasa dziedziczy wszystkie metody klasy Object. Stworzone przez Ciebie klasy dziedziczyły metody, o których w ogóle nie wiedziłeś.

1 equals(Object o)

```
Pies p = new Pies();
Kot k = new Kot();
if (p.equals(k)) {
    System.out.println("prawda");
} else {
    System.out.println("fałsz");
}
```

Wiersz polecenia

T:\>java ObjectTester
fałsz

Informuje, czy dwa obiekty są uważane za „równe” (w dodatku B. wyjaśnimy, co tak naprawdę oznacza ta „równość”).

2 getClass()

```
Kot k = new Kot();
System.out.println(k.getClass());
```

Wiersz polecenia

T:\>java ObjectTester
class Kot

Zwraca informacje o klasie, na podstawie której został utworzony dany obiekt.

3 hashCode()

```
Kot k = new Kot();
System.out.println(k.hashCode());
```

Wiersz polecenia

T:\>java ObjectTester
26399554

Wyświetla kod mieszący obiektu (jak na razie możesz wyobrażać go sobie jako unikalny identyfikator konkretnego obiektu).

4 toString()

```
Kot k = new Kot();
System.out.println(k.toString());
```

Wiersz polecenia

T:\>java ObjectTester
Kot@192d342

Wyświetla komunikat tekstowy zawierający nazwę klasy oraz jakiś numer, który tak naprawdę w ogóle nas nie obchodzi.

Klasa Object i klasy abstrakcyjne

| Nie istniejąca
grupie pytania

P: Czy klasa Object jest klasą abstrakcyjną?

O: Nie. W każdym razie nie z formalnego punktu widzenia. Nie jest to klasa abstrakcyjna, gdyż jej metody zostały zaimplementowane, a inne klasy mogą je dziedziczyć i stosować bez konieczności przesłaniania.

P: Czy zatem można przesłaniać metody tej klasy?

O: Niektóre z nich. Jednak inne zostały sfinalizowane, co oznacza, że przesłanianie ich nie jest możliwe. Jednak zalecane jest (i to bardzo), aby we własnych klasach przesłaniać metody hashCode(), equals() oraz toString(), a w dalszej części książki dowieś się, jak należy to robić. Jednak niektóre metody, takie jak getClass(), muszą działać w konkretny, ściśle określony i zagwarantowany sposób.

P: Skoro metody klasy ArrayList są na tyle ogólne, że mogą operować na obiektach klasy Object, to co daje stosowanie zapisu ArrayList<Portal>? Sądziłem, że ogranicza on możliwości obiektu ArrayList, zezwalając na zapisywanie w nim wyłącznie obiektów Portal?

O: Dokładnie tak. Przed pojawiением się wersji 5.0 w Javie nie można było ograniczyć możliwości obiektów ArrayList. W zasadzie odpowiadały one temu, co teraz, w Javie 5.0, możesz uzyskać przy wykorzystaniu wyrażenia ArrayList<Object>. Innymi słowy, były one **obiektami ArrayList, których możliwości ograniczono do przechowywania obiektów klasy Object**, czyli dowolnych obiektów dowolnego typu! Szczególni tej nowej składni z nawiasami kątowymi zajmiemy się w dalszej części książki.

P: Dobra, wróćmy do klasy Object i do tego, że nie jest to klasa abstrakcyjna (skoro tak, to przypuszczam, że jest to klasa konkretna). Ale JAK można komuś pozwolić na stworzenie obiektu Object? Czy to nie jest równie dziwaczne jak stworzenie obiektu Zwierze?

O: To dobre pytanie! Dlaczego tworzenie nowych obiektów Object jest dopuszczalne? Dlatego, że czasami chcemy używać takich ogólnych obiektów jako... no cóż... obiektów. Lekkich obiektów. Zdecydowanie najczęściej spotykanym zastosowaniem obiektów Object jest synchronizacja wątków (które zajmiemy się w rozdziale 15.). Jak na razie przyjmij do wiadomości, że choć możesz, to raczej sporadycznie będziesz tworzyć obiekty klasy Object.

P: A zatem czy słuszne jest stwierdzenie, że jedynym celem istnienia typu Object jest możliwość wykorzystania go w polimorficznych argumentach i wartościach wynikowych? Takich jak ArrayList?

O: Klasa Object została utworzona z dwóch podstawowych powodów: aby służyć jako polimorficzny typ argumentów w metodach, które muszą operować na obiektach dowolnych klas, oraz by zapewnić normalne metody, których podczas działania programów mogą potrzebować wszystkie obiekty. Niektóre z najważniejszych metod klasy Object są związane z wątkami, poznasz je w dalszej części książki.

P: Jeśli używanie polimorficznych argumentów jest takie super, to czy nie można by używać argumentów i wartości wynikowych typu Object we WSZYSTKICH tworzonych metodach?

O: Ehhh... pomyśl, co by się w takim przypadku stało. Po pierwsze, zniweźlibyś wszystkie zalety kontroli typów — jednego z najważniejszych mechanizmów służących do zapewniania bezpieczeństwa i poprawności kodu. Dzięki kontroli typów Java jest w stanie zagwarantować, że nie będziesz mógł zażądać od obiektu wykonania czynności, którą tak naprawdę chciałeś wykonać, korzystając z innego obiektu. Na przykład poprosić obiekt Ferrari (który uważałeś za Toster), aby zrobić grzankę. Jednak w rzeczywistości, nawet gdybyś wszędzie używał odwołań typu Object, to i tak taki przypadek nie miałby miejsca. Jeśli bowiem do obiektów odwołujemy się za pośrednictwem odwołania typu Object, to Java uważa, że odwołujemy się do obiektu Object. A to oznacza, że jedynymi metodami, jakie można wywoływać, posługując się takim odwołaniem, są metody zadeklarowane w klasie Object! A zatem gdybyś użył poniższego fragmentu kodu:

```
Object o = new Ferrari();  
o.zrobGrzanke();  
// niedozwolone!!
```

to nie byłbyś w stanie nawet go skompilować.

Java jest językiem o rygorystycznej kontroli typów, dlatego też kompilator sprawdza, czy odwołania do obiektów, których używamy do wywoływania metod, potrafią na te żądania odpowiedzieć. Innymi słowy, metodę można wywołać, posługując się odwołaniem do obiektu **wyłącznie** w przypadku, gdy typ tego odwołania **udostępnia** daną metodę. Zagadnieniem tym zajmiemy się bardziej szczegółowo w dalszej części książki, a zatem nie przejmuj się, jeśli nie do końca to zrozumiałeś.

Stosowanie polimorficznych odwołań typu Object ma swoją cenę...

Zanim zaczniesz używać klasy Object jako ultra-elastycznego typu wszystkich argumentów i wartości wynikowych, powinieneś dowiedzieć się o jeszcze jednym niewielkim problemie związanym ze stosowaniem odwołań typu Object. Pamiętaj, że nie chodzi nam tu o tworzenie obiektów klasy Object — mamy na myśli tworzenie obiektów jakiegoś innego typu oraz stosowanie odwołań typu Object.

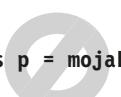
Kiedy zapisujesz obiekt na liście `ListArray<Pies>`, to zapisujesz na niej obiekt klasy Pies i później odczytyujesz obiekt tego samego typu.

```
ArrayList<Pies> mojaListaPsow = new ArrayList<Pies>(); ← Tworzymy obiekt ArrayList, w którym będzie można przechowywać obiekty klasy Pies.  
Pies mojPies = new Pies(); ← Tworzymy obiekt Pies.  
mojaListaPsow.add(mojPies); ← Dodajemy go do tablicy.  
Pies p = mojaListaPsow.get(0); ← Pobieramy element z tablicy i zapisujemy go w nowej zmiennej typu Pies. (Ponieważ lista została utworzona jako ArrayList<Pies>, zatem możesz sobie wyobrazić, że metoda get() deklarowana wartość wynikową typu Pies).
```

Co się jednak stanie, jeśli tablicę zadeklarujesz jako `ArrayList<Object>`? Jeśli chcesz utworzyć tablicę zdolną do przechowywania *dowolnych* obiektów, możesz ją zadeklarować w następujący sposób:

```
ArrayList<Object> mojaListaPsow = new ArrayList<Object>(); ← Tworzymy obiekt ArrayList, w którym będzie można przechowywać dowolne obiekty — typu Object.  
Pies mojPies = new Pies(); ← Tworzymy obiekt Pies.  
mojaListaPsow.add(mojPies); ← Dodajemy go do tablicy. } (Te dwa kroki są takie same jak wcześniej.)
```

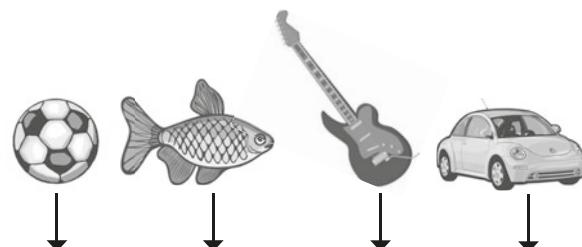
Co się jednak dzieje, kiedy spróbujesz pobrać ten obiekt z tablicy?


`Pies p = mojaListaPsow.get(0);`

NIE! Tego się nie uda skompilować! W przypadku utworzenia tablicy jako `ArrayList<Object>` metoda `get()` będzie zwracać wyniki typu Object. Kompilator wie zatem, że obiekty dziedziczą po klasie Object (pistożonej gdzieś w drzewie dziedziczenia) — nie wie natomiast, że są to obiekty Pies!

Wszystkie obiekty odczytywane z `ArrayList<Object>` są obiektami klasy Object, niezależnie od tego, jakiej klasy faktycznie jest odczytywany obiekt ani jakiego typu było odwołanie do niego, zastosowane podczas dodawania obiektu do tablicy.

Obiekty są zapisywane jako Pilka, Ryba, Gitara i Auto.



Jednak podczas odczytywania stają się obiektami klasy Object.

Obiekty odczytywane z `ArrayList<Object>` zachowują się tak, jak gdyby były obiektami ogólnej klasy Object. Kompilator nie może założyć, że obiekt odczytywany z listy będzie miał jakikolwiek inny typ niż Object.

Kiedy Pies nie zachowuje się jak Pies

Problem związany z traktowaniem wszystkich obiektów w sposób polimorficzny — czyli jako obiektów klasy Object — polega na tym, że obiekty takie *wydają się* zapominać swoją prawdziwą tożsamość (na szczęście nie jest to trwała amnezja). *Pies zdaje się tracić swój psi charakter*. Sprawdźmy, co się stanie, kiedy zapiszemy obiekt Pies w tablicy ArrayList i spróbujemy go z niej odczytać.



Nie wiem, co on do mnie mówi. Siad? Aport?
Leżeć? Hmm... Nie przypominam sobie żadnego z tych poleceń.

```
public void dalej() {
    Pies xPies = new Pies();
    Pies jakisPies = getObject(xPies); ←
}

public Object getObject(Object o) {
    return o;
}
```

Zwracamy odwołanie do tego samego obiektu Pies, jednak jest ono typu Object. Jest to rozwiązanie całkowicie poprawne. Uwaga: podobnie działa metoda `get()`, w przypadku gdy tablicę `ArrayList` zadeklarujemy jako `ArrayList<Object>`, a nie `ArrayList<Pies>`.

W tym wierszu jest błąd! Nawet gdyby metoda zwracająca odwołanie do dokładnie tego samego obiektu Pies, do którego odwoływać się argument, to zastosowanie wartości wynikowej typu Object oznacza, że kompilator nie pozwoli przypisać wartości wynikowej metody do zmiennej jakiegokolwiek innego typu niż Object.

Wiersz polecenia

```
T:>javac PulaPsowTest.java
PulaPsowTest.java:22: incompatible types
found   : java.lang.Object
required: Pies
        Pies jakisPies = getObject(xPies);
                                         ^
1 error
T:>_
```

Kompilator nie wie, że obiekt zwracany przez metodę w rzeczywistości jest obiektem Pies, dlatego też nie pozwoli zapisać go w odwołaniu typu Pies. (Na następnej stronie przekonasz się, dlaczego tak się dzieje).

DOBRZE

```
public void dalej() {
    Pies mojPies = new Pies();
    Object jakisPies = getObject(mojPies); ←
}

public Object getObject(Object o) {
    return o;
}
```

Ta wersja kodu działa (choć, jak się wkrótce przekonasz, nie jest szczególnie użyteczna), gdyż do odwołania typu Object można przypisać WSZYSTKO. Wynika to z faktu, iż każda klasa spełnia test relacji JEST z klasą Object. Każdy obiekt w Javie jest obiektem klasy Object, gdyż w Javie klasa Object znajduje się na samym szczycie drzewa dziedziczenia każdej klasy.

Object nie szczeka

Zatem już wiemy, że jeśli do obiektu odwołuje się zmienna zadeklarowana jako zmienna typu `Object`, to jej zawartości nie można przypisać innej zmiennej jakiejś innej klasy. Wiemy także, że taka sytuacja może się zdarzyć, jeśli wartość wynikowa metody lub argument zostanie zadeklarowany jako typu `Object`; a tak właśnie by było w przypadku umieszczenia obiektu w tablicy `ArrayList` zawierającej obiekty `Object` (`ArrayList<Object>`). Jakie są jednak tego implikacje? Czy to jakiś problem używać odwołań typu `Object` do obiektu klasy `Pies`? Spróbujmy wywołać jakąś metodę obiektu `Pies`. Który-Kompilator-Uważa-Za-Obiekt-Object.

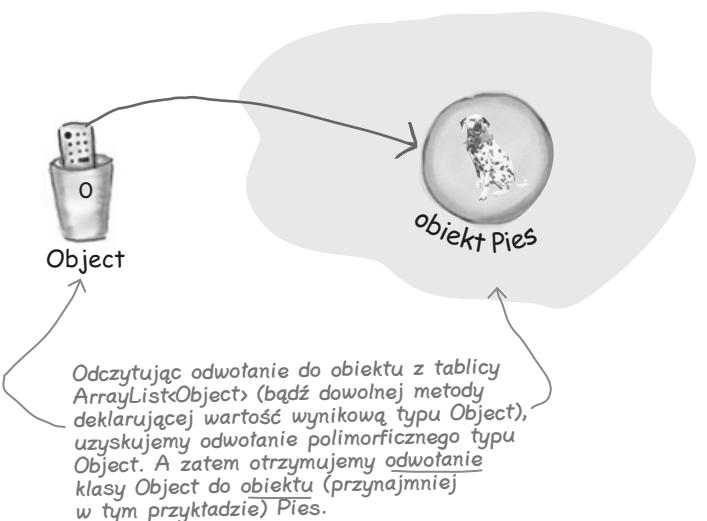
```
Object o = al.get(index);
int i = o.hashCode();
o.szczekaj();
```

To się nie skompiluje! ↗

Kompilator decyduje o możliwości wywołania metody na podstawie typu odwołania, a nie na podstawie typu faktycznego obiektu.

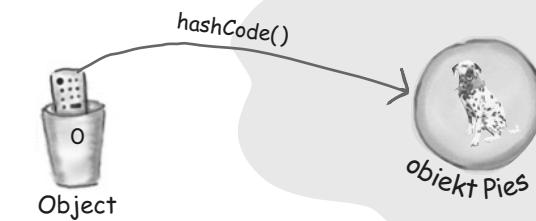
Choć wiesz, jakie są prawdziwe możliwości obiektu (ale to naprawdę *jest Pies*, słowo honoru...), to kompilator uważa go za obiekt ogólnej klasy `Object`. Ponieważ kompilator wie tylko tyle, że w tablicy zapisałeś obiekt `Button` albo obiekt `KuchenkaMikrofalowa`, albo jeszcze jakiś inny obiekt, który zupełnie nie wie, jak należy szczekać.

Kompilator sprawdza klasę *odwołania* — a nie klasę *obiektu* — i na tej podstawie decyduje, czy danego odwołania można użyć do wywołania metody.



To jest w porządku. Klasa `Object` dysponuje metodą `hashCode()`, a zatem można ją wywołać dla dowolnego obiektu w Javie.

Nie możesz tego zrobić!! Obiekt klasy `Object` nie ma najmniejszego pojęcia, co może robić metoda `szczerkaj()`. Choć TY wiesz, że w danym miejscu tablicy jest przechowywany obiekt `Pies`, kompilator o tym nie wie.



Object
equals()
getClass()
hashCode()
toString()

Metoda, którą wywołujesz, używając odwołania, MUSI być dostępna w klasie określonej w deklaracji odwołania. Faktyczny typ obiektu nie ma w tym przypadku żadnego znaczenia.

`o.hashCode();`

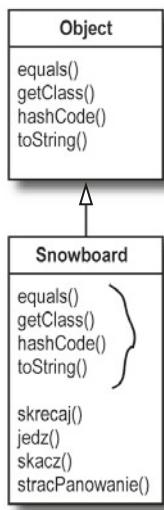
Odwołanie „`o`” zostało zadeklarowane jako odwołanie klasy `Object`, a zatem, postępując się nim, możesz wywoływać metody wyłącznie, jeśli są dostępne w klasie `Object`.



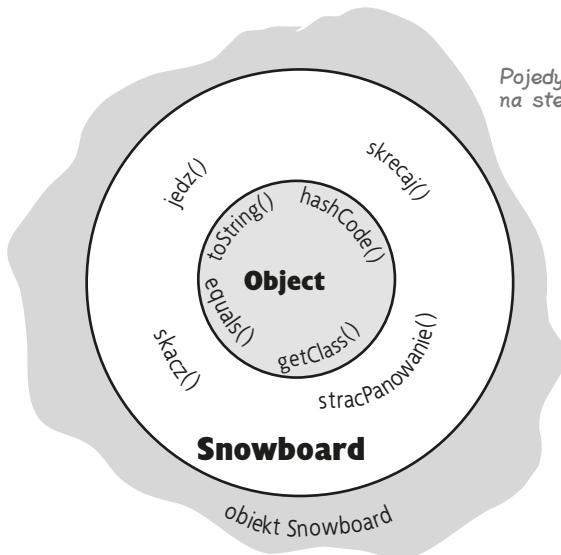
Traktuje mnie jak Object. A przecież potrafię znacznie więcej... gdyby tylko chciał zobaczyć mnie taką, jaką naprawdę jestem.

Połącz się ze swoim wewnętrznym Object-em

Obiekt zawiera *wszystko*, co odziedziczył po swoich klasach bazowych. A to oznacza, że *każdy* obiekt — niezależnie od jego bieżącego typu — jest także obiektem klasy Object. A zatem każdy obiekt w Javie może być traktowany nie tylko jako Pies, Snowboard bądź Piekarz, lecz także jako Object. Wykonując instrukcję `new Snowboard()`, na stercie tworzony jest jeden obiekt — obiekt Snowboard — jednak jest on „otoczką” wokół „jądra” reprezentującego obiekt Object.



Snowboard dziedziczy metody po klasie bazowej Object i dodaje cztery własne.



Na stercie znajduje się tylko JEDEN obiekt, obiekt Snowboard. Jednak obiekt ten składa się zarówno z części unikalnej dla klasy Snowboard, jak i z części charakterystycznej dla klasy Object.

„Polimorfizm” oznacza „wiele form”.

Snobaord możesz traktować zarówno jako Snowboard, jak i jako Object.

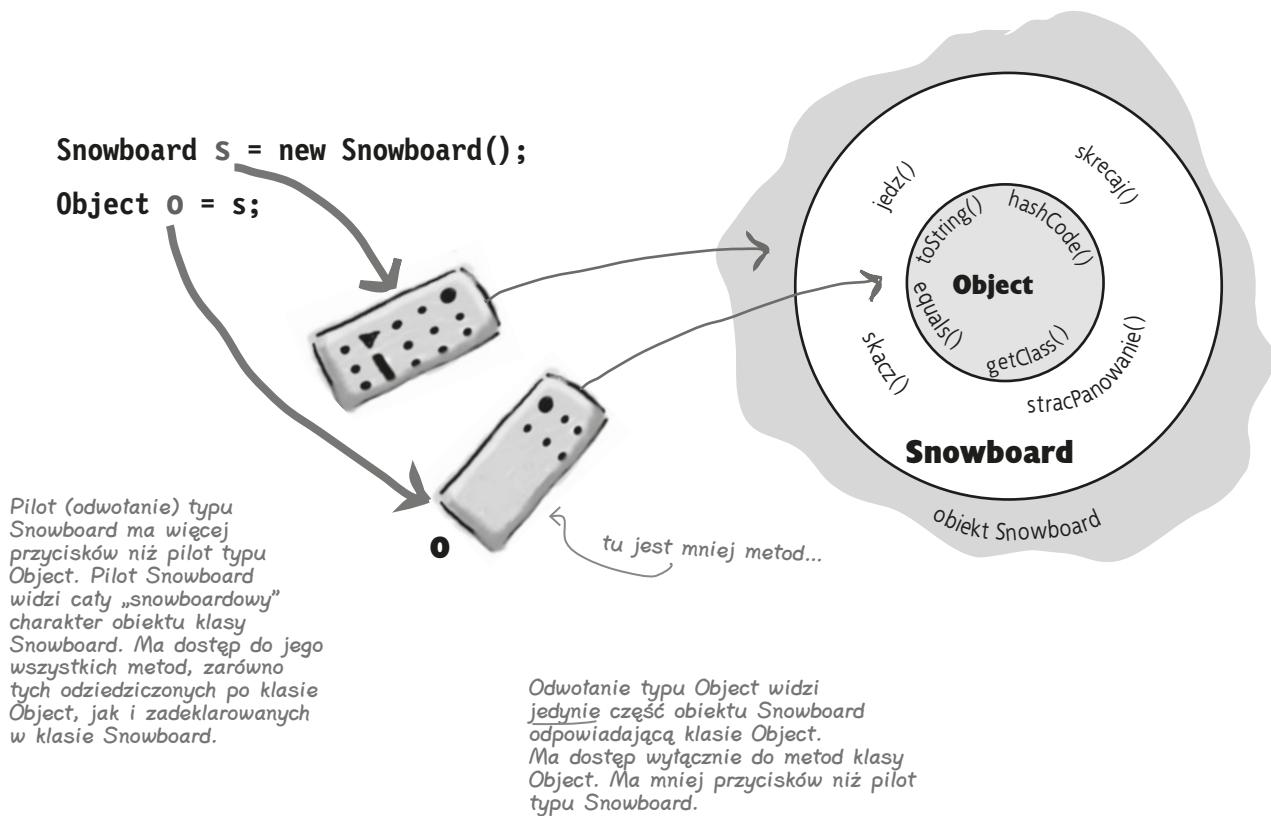
Jeśli odwołanie do obiektu porównamy do pilota, to schodząc w dół drzewa dziedziczenia na takim pilocie będzie się pojawiać coraz więcej przycisków. Pilot (odwołanie) typu Object ma niewiele przycisków — odpowiadają one dostępnym metodom klasy Object. Jednak pilot typu Snowboard ma zarówno wszystkie przyciski klasy Object, jak również kilka nowych przycisków (odpowiadających nowym metodom) klasy Snowboard. Im bardziej szczegółowa i wyspecjalizowana jest klasa, tym więcej przycisków ma pilot.

Oczywiście nie zawsze tak musi być — klasa potomna wcale nie musi dodawać żadnych nowych metod, a jedynie przesłaniać metody klasy bazowej. Najważniejsze jest jednak to, że nawet jeśli obiekt będzie typu Snowboard, to odwołanie Object do obiektu typu Snowboard nie będzie udostępniać metod charakterystycznych dla klasy Snowboard.

Jeśli umieszczasz obiekt w tablicy `ArrayList<Object>`, to możesz go traktować wyłącznie jako Object, niezależnie od jego początkowego typu.

Odczytując odwołanie z tablicy `ArrayList<Object>` zawsze jest to odwołanie typu Object.

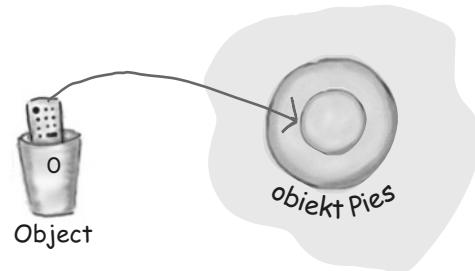
Oznacza to, że uzyskujesz pilot typu Object.



Rzutowanie obiektów



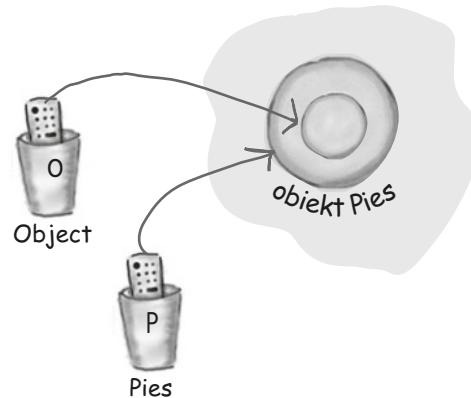
Rzutowanie odwołania do obiektu na faktyczny typ tego obiektu.



To wciąż jest *obiekt Pies*, jednak aby wywoływać metody charakterystyczne dla tej klasy, konieczne jest *odwołanie* klasy `Pies`. Jeśli jesteś *pewien**, że obiekt faktycznie jest klasy `Pies`, to możesz utworzyć nowe odwołanie do niego — odwołanie klasy `Pies` — kopiując odwołanie typu `Object` i zapisując je w zmiennej referencyjnej klasy `Pies`, wykorzystując przy tym rzutowanie — (`Pies`). Odwołanie utworzone w taki sposób można wykorzystać do wywoływania metod klasy `Pies`.

```
Object o = lista.get(index);
```

```
Pies p = (Pies) o; ← Rzutujemy obiekt klasy Object  
p.szczekaj(); ← z powrotem na klasę Pies, gdyż  
wiemy, że obiekt jest tej klasy.
```



*Jeśli *nie* jesteś pewien, czy to naprawdę jest obiekt klasy `Pies`, możesz to sprawdzić przy użyciu operatora `instanceof`. Jeśli wykonując rzutowanie, pomyliś się, to podczas wykonywania programu zostanie zgłoszony wyjątek `ClassCastException`, a realizacja programu zostanie przerwana.

```
if (o instanceof Pies)  
    Pies p = (Pies) o;  
}
```

Teraz widziałeś już, jak wiele uwagi Java zwraca na metody dostępne w klasie odwołania.

Metodę obiektu można wywołać tylko wtedy, gdy jest ona dostępna także w klasie zmiennej referencyjnej.

Wyobraź sobie, że publiczne metody Twojej klasy stanowią kontrakt, obietnicę składaną zewnętrznemu światu i określającą, co obiekty danej klasy potrafią robić.



Pisząc klasę, niemal zawsze tworzymy jakieś metody, które będą dostępne dla zewnętrznego kodu. W przypadku metod zapewnienie ich *dostępności* zazwyczaj sprawdza się do oznaczenia ich jako publiczne.

Wyobraź sobie następującą sytuację: piszesz kod dla programu zarządzania księgowością niewielkiej firmy — niestandardową aplikację dla „Sklepu Surfowego Simona”. Ponieważ bardzo lubisz wielokrotnie wykorzystywać kod, znalazłeś klasę Rachunek, która wydaje się idealnie spełniać Twoje wymagania, przynajmniej tak można by sądzić na podstawie jej dokumentacji. Każdy obiekt tej klasy reprezentuje rachunek, który poszczególni klienci mają w sklepie. I już wyobrażałeś sobie prowadzenie działalności poprzez wywołanie metod `wplata()` i `obcielenie()` dla obiektów poszczególnych rachunków, kiedy nagle uświadomiłeś sobie, że potrzebna Ci będzie metoda do wyznaczania salda rachunku. Nie ma problemu — jest metoda `wyznaczSaldo()`, która doskonale powinna sobie z tym zadaniem poradzić.

Rachunek
obcielenie(double kwt)
wplata(double kwt)
double wyznaczSaldo()

Jest jednak mały problem... kiedy wywołujesz tę metodę, podczas działania programu jest zgłoszony wyjątek, który doprowadza do przerwania działania całego programu. I zapomnij o dokumentacji — według niej klasa nie ma takiej metody. To problem!

Jednak taka sytuacja nie może Ci się przydarzyć, gdyż za każdym razem, gdy używasz odwołania i operatora kropki (na przykład: `a.zrobCos()`), kompilator analizuje typ *odwołania* (czyli typ, jaki został zadeklarowany podczas tworzenia zmiennej „a”) i sprawdza, czy klasa gwarantuje istnienie metody, czy metoda akceptuje argumenty, które umieściłeś w jej wywołaniu oraz czy zwraca wartość, której oczekujesz.

Pamiętaj, że kompilator sprawdza typ *odwołania*, a nie typ faktycznego obiektu, na który to odwołanie wskazuje.

A co, jeśli musimy zmienić kontrakt?

W porządku. Założmy, że jesteś obiektem Pies. Twoja klasa Pies nie jest *jedynym* kontraktem określającym, kim jesteś. Pamiętaj, że dziedzicysz wszystkie dostępne klasy (co zazwyczaj oznacza klasy *publiczne*) ze wszystkich klas bazowych.

To prawda, że klasa Pies definiuje kontrakt.

Jednak nie jest to *cały* kontrakt.

W skład tego kontraktu wchodzi także całość klasy Psowate.

Jak również całość klasy Zwierze.

Jak również całość klasy Object.

Zgodnie z testem relacji JEST, jesteś obiektem każdej z tych klas — obiektem Psowate, Zwierze oraz Object.

Co się jednak stanie, jeśli osoba która zaprojektowała klasę Pies myślała o programie do symulacji zwierząt, a teraz chce wykorzystać tę samą klasę na jarmarku naukowym poświęconym obiektom symulującym zwierzęta?

Cóż... w porządku, prawdopodobnie sobie poradzisz.

A co się stanie, jeśli potem będzie chciała wykorzystać tę samą klasę do programu obsługi Zwierzaków Domowych? *Nie dysponujesz żadnym zachowaniem charakterystycznym dla zwierzaka domowego.* ZwierzakDomowy musi mieć takie metody jak badzMilutki() i bawSie().

No dobrze, a teraz założmy, że jesteś twórcą klasy Pies. Nie ma problemu, nieprawdaż? Wystarczy dodać do klasy Pies kilka metod. Dodanie metod nie powinno wywołać żadnych problemów w kodzie innych osób, gdyż nie wiąże się z modyfikacją istniejących metod, które inne osoby mogłyby wywoływać korzystając z obiektów Pies.

Czy zauważasz jakiekolwiek wady takiego rozwiązania (czyli dodanie do klasy Pies metod klasy ZwierzakDomowy)?



WYSIL SZARE KOMÓRKI

Pomyśl, co TY byś zrobił, gdybyś był twórcą klasy Pies i musiał zmodyfikować ją w taki sposób, aby mogła wykonywać czynności charakterystyczne dla klasy ZwierzakDomowy. Wiemy, że rozwiązanie polegające na dodaniu nowych czynności (metod) do klasy Pies zda egzamin i nie przysporzy żadnych problemów w istniejącym kodzie wykorzystującym tę klasę.

Ale... to jest program symulujący sklep ze zwierzętami domowymi. Jest w nim używanych znacznie więcej obiektów, a nie tylko obiekty Pies! A co, jeśli ktoś będzie chciał użyć Twojej klasy w programie, w którym pojawiają się *dzikie psy*? Jak myślisz, jakie mogłyby być rozwiązania tego problemu? I nie przejmuj się tym, w jaki sposób działa Java — po prostu postaraj się wyobrazić sobie, w jaki sposób *chciałbyś* rozwiązać problem modyfikacji niektórych spośród klas symulujących zwierzęta i dodania do nich zachowań zwierzaków domowych.

Przerwij na chwilę lekturę i zastanów się nad tym, **zanim przejdziesz na następną stronę**, na której zaczniemy wszystko wyjaśniać.

(gdyż w takim przypadku całe ćwiczenie będzie całkowicie bezużyteczne, a Ty stracisz jedyną, wielką okazję do potrenowania swojego mózgu i zmuszenia go do spalenia kilku kalorii).

Rozważmy różne możliwości projektowe, jakie można zastosować w celu wykorzystania niektórych z naszych istniejących klas w programie symulacji sklepu ze zwierzętami.

Na kilku następnych stronach przedstawimy niektóre możliwe rozwiązania. Jak na razie w ogóle nie przejmujemy się tym, czy Java jest w stanie zrealizować to, co wymyślimy. Z tym problemem zmierzymy się później — kiedy dokładnie poznamy zalety i wady różnych rozwiązań.

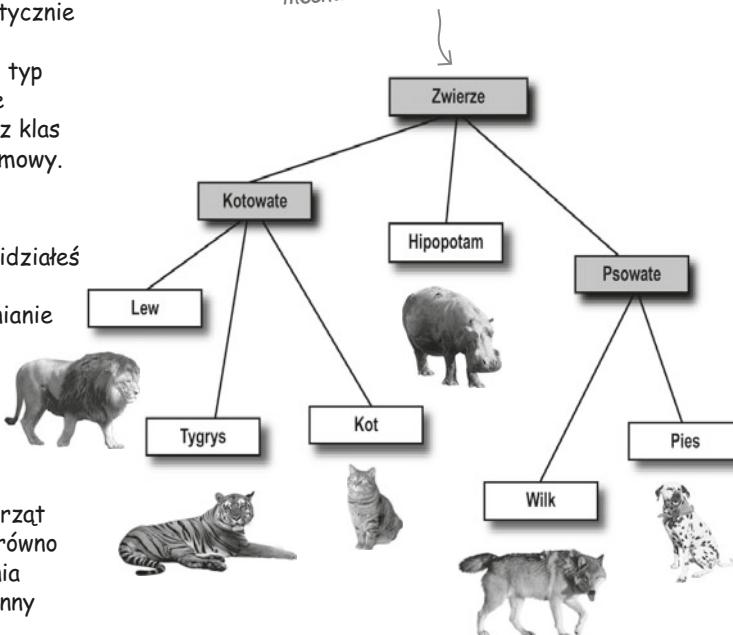
1 Rozwiązanie pierwsze

Zastosujemy najłatwiejsze rozwiązanie i umieścimy metody klasy ZwierzakDomowy w klasie Zwierze.

Zalety:

Wszystkie klasy potomne klasy Zwierze od razu dziedziczą zachowania klasy ZwierzakDomowy. Nie musimy w żaden sposób modyfikować istniejących klas potomnych klasy Zwierze, a wszystkie takie klasy utworzone w przyszłości automatycznie dziedziczą te metody. W ten sposób klasa Zwierze może być używana jako typ polimorficzny w programie, który chce traktować obiekty klasy Zwierze (oraz klas potomnych) jako obiekty ZwierzakDomowy.

Umieszczamy kod wszystkich metod zwierzaków domowych w klasie bazowej, aby był dostępny dzięki mechanizmom dziedziczenia.



Wady:

Dobra... zatem kiedy po raz ostatni widziałeś hipopotama w sklepie ze zwierzętami domowymi? A lwa? A wilka? Udostępnianie w tych obiektach metod innych niż te charakterystyczne dla zwierząt nieudomowionych mogłoby być niebezpieczne.

Poza tym niemal na pewno będziemy musieli zmodyfikować takie klasy zwierząt domowych jak Pies i Kot, ponieważ zarówno psy, jak i koty implementują zachowania zwierząt domowych w CAŁKOWICIE inny sposób (przynajmniej u nas w domu).

2 Rozwiążanie drugie

Rozpoczynamy podobnie jak w rozwiążaniu pierwszym – czyli umieszczamy metody klasy ZwierakDomowy w klasie Zwierze – lecz deklarujemy je jako metody abstrakcyjne, przez co klasy potomne będą musiały je przesłonić.

Zalety:

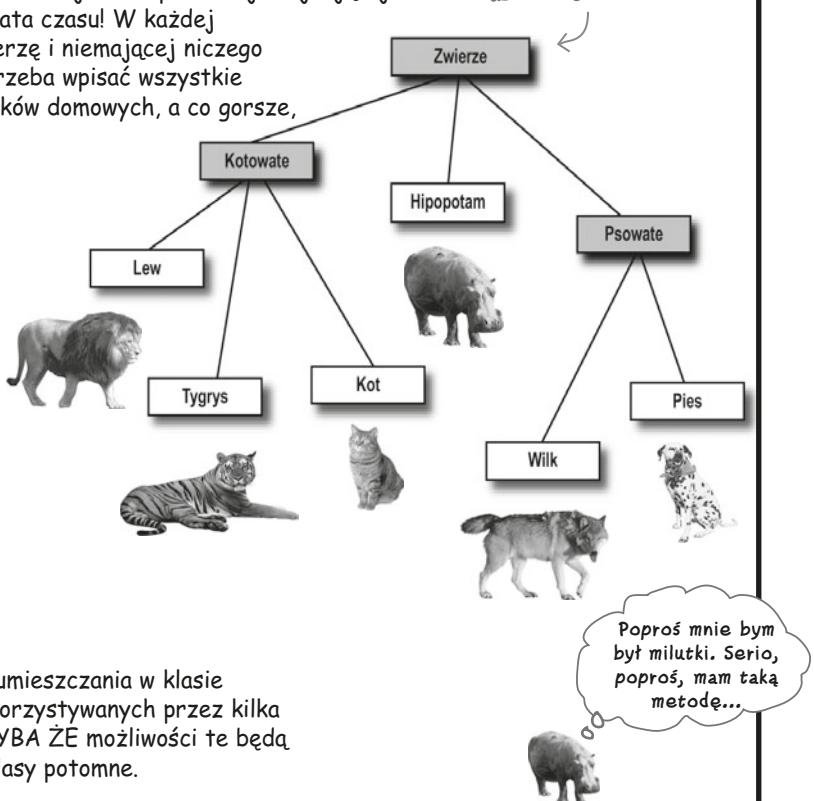
Takie rozwiążanie zapewni nam wszystkie korzyści rozwiązania pierwszego, a jednocześnie pozwoli uniknąć sytuacji, w której zwierzęta nieudomowione będą mogły korzystać z metod charakterystycznych dla zwierząt domowych (takich jak badzMilutki()). Wszystkie klasy zwierząt dysponowałyby tymi metodami (gdyż zostały one umieszczone w klasie bazowej Zwierze), jednak nie dziedziczyłyby żadnych możliwości funkcjonalnych, ponieważ są to metody abstrakcyjne. Oczywiście wszystkie klasy MUSZĄ przesłaniać te metody, jednak same metody mogą nie wykonywać żadnych czynności..

Wady:

Ponieważ metody zwierząt domowych umieszczone w klasie Zwierze są metodami abstrakcyjnymi, zatem konkretne klasy potomne klasy Zwierze muszą je implementować. (Pamiętaj, że wszystkie metody abstrakcyjne muszą być zaimplementowane w pierwszej konkretnej klasie potomnej znajdującej się w drzewie dziedziczenia). Co za strata czasu! W każdej konkretnej klasie reprezentującej zwierzę i niemającej niczego wspólnego ze zwierzątkami domowymi trzeba wpisać wszystkie metody charakterystyczne dla zwierząt domowych, a co gorsze, trzeba je będzie wpisywać także we wszystkich klasach tworzonych w przyszłości. I chociaż rozwiązuje to problem nieudomowionych zwierząt WYKONUJĄCYCH czynności typowe dla zwierząt domowych (co byłoby możliwe, gdyby klasy te dziedziczyły odpowiednie możliwości funkcjonalne po klasie Zwierze), to jednak taki kontrakt też nie jest dobry. Wszystkie klasy potomne klasy Zwierze, nawet te, które nie reprezentują zwierząt udomowionych, ogłosząby światu, że dysponują metodami zwierząt domowych, choć w rzeczywistości, w ich przypadku, metody te nie WYKONUJĄ żadnych operacji.

To nie jest dobre rozwiązanie. Pomyśl umieszczania w klasie Zwierze możliwości funkcjonalnych wykorzystywanych przez kilka klas potomnych wydaje się błędem; CHYBA ŻE możliwości te będą wykorzystywane przez WSZYSTKIE klasy potomne.

Umieszczamy wszystkie metody zwierząt domowych w klasie bazowej, jednak nie implementujemy ich. Wszystkie te metody deklarujemy jako abstrakcyjne.



3 Rozwiążanie trzecie

Umieszczamy metody charakterystyczne dla zwierzaków domowych **WYŁĄCZNIE** w klasach, w jakich powinny się znaleźć.

Zalety:

Nie musimy się już obawiać, że Hipopotam przywita nas w drzwiach lub będzie lizać po twarzy. Metody znajdują się w klasach, w których powinny się znaleźć i w **ŻADNYCH** innych. Metody typowe dla zwierzaków domowych mogą być zaimplementowane w klasach Pies i Kot, natomiast żadne pozostałe klasy nie muszą nic o nich wiedzieć.

Wady:

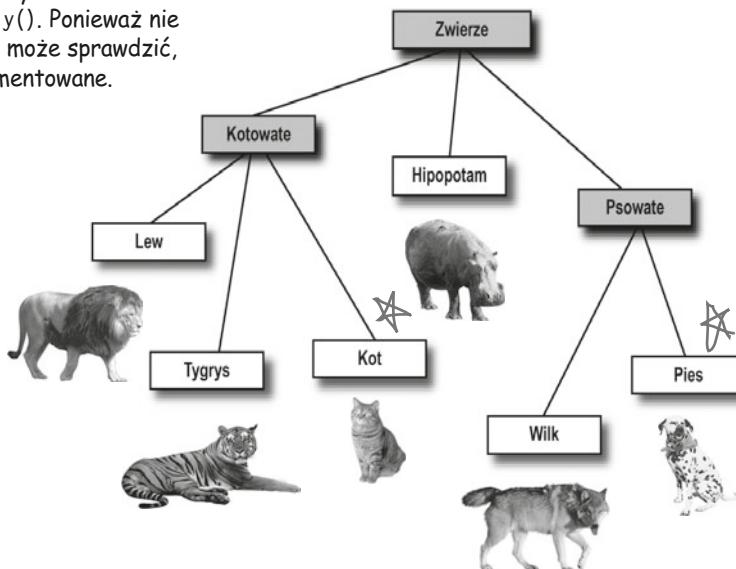
Rozwiążanie to przysparza dwóch poważnych problemów. Po pierwsze trzeba by określić jakiś protokół, a wszyscy twórcy klas reprezentujących zwierzęta domowe musieliby ZNAĆ ten protokół, i to zarówno teraz, jak i w przyszłości. Przez protokół rozumiemy konkretne metody, które wszystkie zwierzaki domowe powinny udostępniać (kontrakt zwierzaka domowego, bez czegokolwiek, co mogłoby go wspomóc). Co się jednak stanie, gdy jeden z programistów popełni niewielki błąd podczas implementacji tego kontraktu? Na przykład pomyli typ i zamiast metody, która pobiera argument typu String, stworzy metodę, która pobiera liczbę całkowitą. Albo jeśli pomyli nazwę metody i zamiast badzMilutki() stworzy metodę badzMily(). Ponieważ nie jest to kontrakt, zatem kompilator nie może sprawdzić, czy metody zostały poprawnie zaimplementowane.

Bez trudu można sobie wyobrazić sytuację, w której ktoś spróbowałby użyć klas zwierząt domowych i okazałoby się, że nie wszystkie z nich działają poprawnie.

Po drugie, metody zwierząt domowych nie wykorzystują polymorfizmu. Każda klasa, w której zachowania zwierząt domowych są stosowane, musiałaby znać wszystkie pozostałe klasy! Innymi słowy, w tym przypadku nie można zastosować klasy Zwierze jako typu polymorficznego, gdyż kompilator nie pozwoli na wywoływanie metod charakterystycznych dla zwierząt domowych za pośrednictwem odwołań klasy Zwierze (nawet jeśli w rzeczywistości odwołanie to wskazuje na obiekt Pies). Wynika to z faktu, że klasa Zwierze nie udostępnia tych metod.



Metody charakterystyczne dla zwierząt domowych umieszczany **WYŁĄCZNIE** w klasach zwierząt, które mogą być trzymane w domu, a nie w klasie bazowej Zwierze.

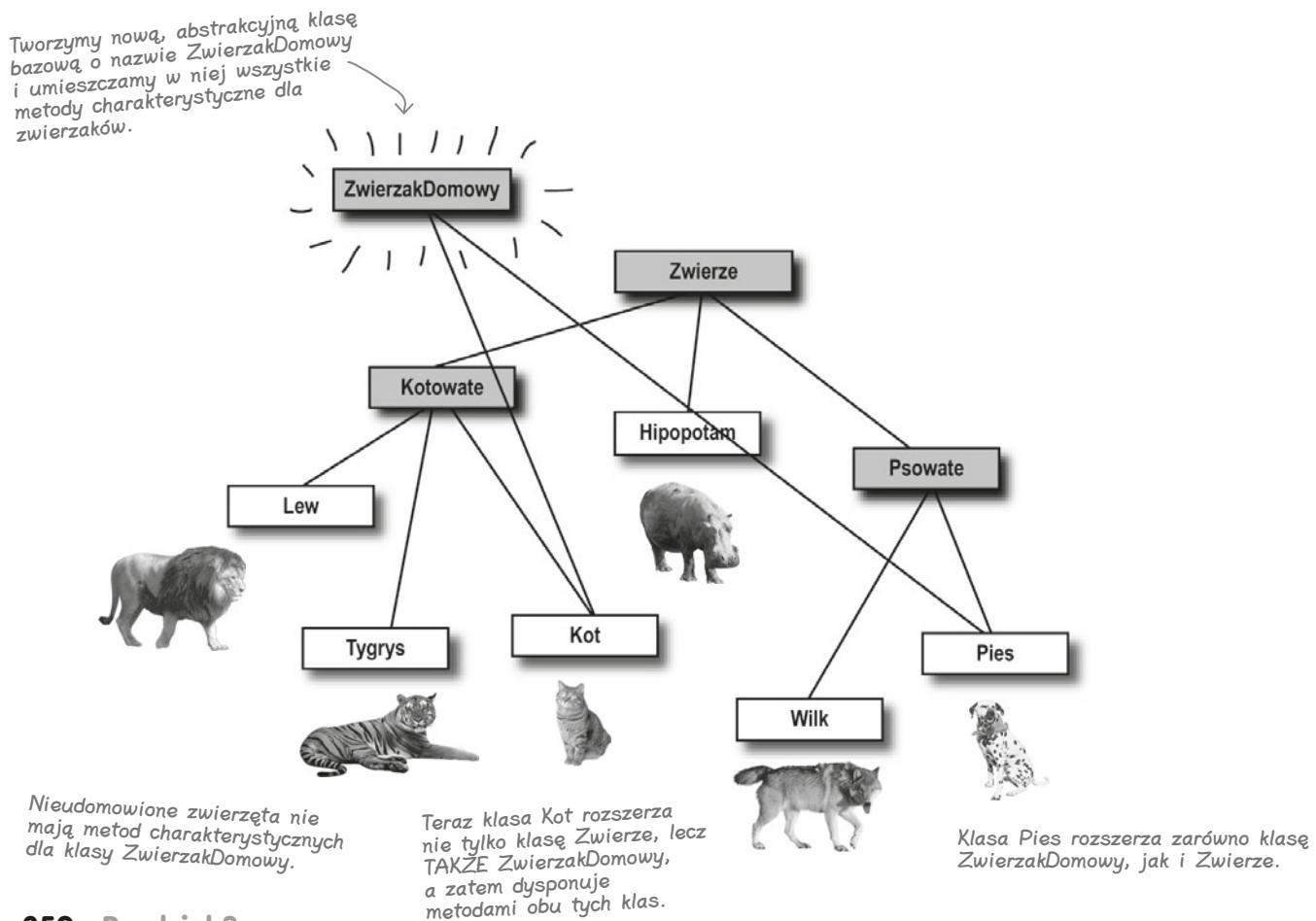


Wielokrotne dziedziczenie?

A zatem, oto czego nam tak NAPRAWDĘ potrzeba:

- Możliwości umieszczenia zachowań charakterystycznych dla zwierzaków domowych tylko w odpowiednich klasach.
- Sposobu, który pozwoli zagwarantować, że we wszystkich klasach zwierzaków domowych zostaną zdefiniowane wszystkie wymagane metody (z takimi samymi argumentami i wartościami wynikowymi) i to bez pukania w niemalowane drewno i cichej nadziei, że wszystkim programistom nasze rozwiązanie będzie działać.
- Sposobu, który pozwoliłby nam na wykorzystanie polimorfizmu, dzięki któremu dla wszystkich obiektów zwierzaków domowych można będzie wywoływać charakterystyczne dla nich metody i to bez konieczności używania argumentów, typów wynikowych i tablic dla każdej z klas zwierzaków.

Wygląda na to, że na samej górze hierarchii dziedziczenia potrzebujemy DWÓCH klas



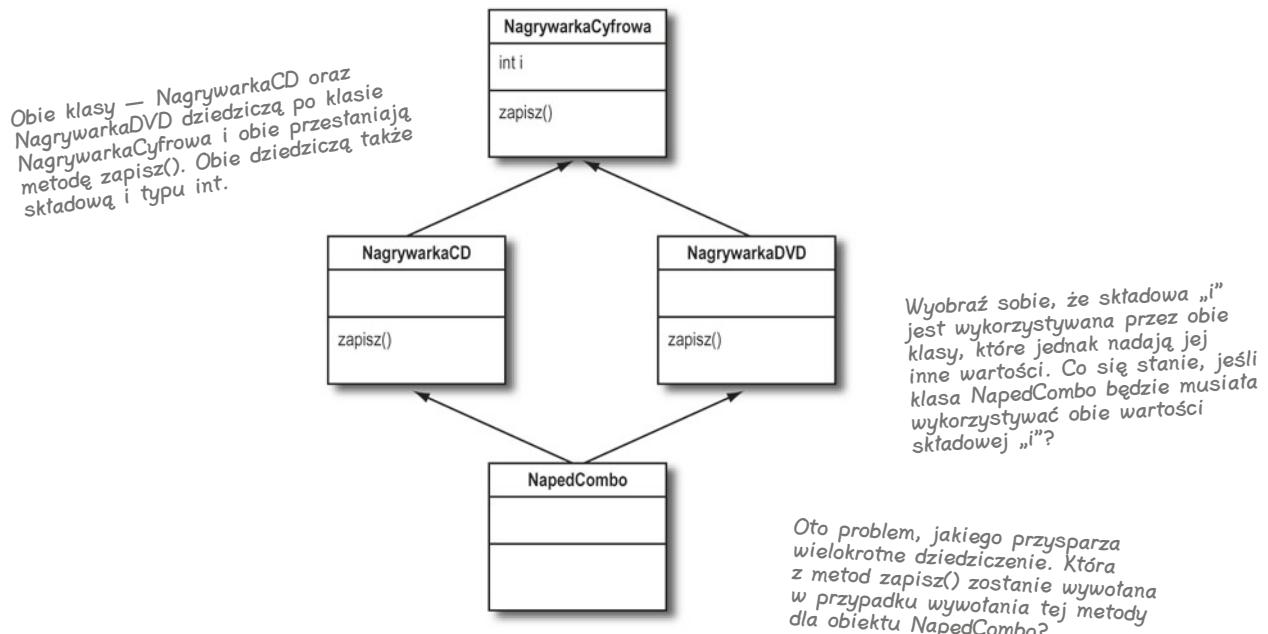
Rozwiążanie wykorzystujące „dwie klasy bazowe” przysparza tylko jednego problemu...

Nosi ono nazwę „wielokrotnego dziedziczenia” i może być naprawdę niebezpieczne.

To znaczy — mogłoby być — gdyby tylko można je było wykorzystać w Javie.

Jednak nie jest, gdyż wielokrotne dziedziczenie może doprowadzić do powstania problemu nazywanego „śmiertelnym rombem”.

Śmiertelny romb



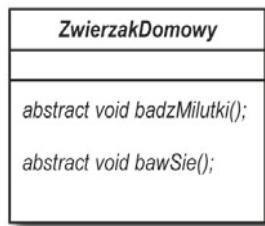
Język, który pozwala na pojawianie się problemu „śmiertelnego rombu”, może doprowadzać do niepotrzebnych komplikacji, gdyż konieczne są dodatkowe reguły rozwiązywania potencjalnych niejasności. Dodatkowe reguły oznaczają dodatkową pracę konieczną zarówno do ich *poznania*, jak i do zwieracania uwagi na „szczególne przypadki”. Java z założenia ma być językiem *prostym*, mającym spójne reguły, które nie są niepotrzebnie rozbudowywane w szczególnych przypadkach. I dlatego (w odróżnieniu od C++) zabezpiecza programistów i sprawia, że nie muszą myśleć o problemie „śmiertelnego rombu”. Jednak właśnie dlatego wracamy do punktu wyjścia! *Jak stworzyć klasę dziedziczącą po klasach Zwierze i ZwierzakDomowy?*

Na pomoc spieszą interfejsy!

Java udostępnia rozwiązanie tego problemu. Są nim *interfejsy*. I nie są to bynajmniej graficzne interfejsy użytkownika ani interfejsy w ogólnym znaczeniu tego słowa, jak na przykład w zdaniu: „Oto publiczny interfejs klasy Button”. Chodzi o **interface** — słowo *kluczowe* Javy.

Interfejsy Javy rozwiążają problem wielokrotnego dziedziczenia, gdyż zapewniają znaczną część korzyści, jakie w przypadku stosowania wielokrotnego dziedziczenia daje możliwość wykorzystywania polimorfizmu, a jednocześnie chroni nas przed problemem „śmiertelnego rombu”.

Interfejsy strzegą nas przed problemem „śmiertelnego rombu” w zadziwiająco prosty sposób — **wymuszają, by wszystkie metody były abstrakcyjne!** Dzięki temu klasa potomna **musi** zaimplementować wszystkie metody (pamiętaj, że wszystkie metody abstrakcyjne muszą zostać zaimplementowane w pierwszej konkretnej klasie potomnej), a podczas działania programu wirtualna maszyna Javy nie ma problemu z wybieraniem, *która* z dwóch odziedziczonych wersji metody należy wywołać.



Interfejs przypomina całkowicie abstrakcyjną klasę.

Wszystkie metody wchodzące w skład interfejsu są abstrakcyjne, zatem wszystkie klasy, które spełniają relację JEST z interfejsem *ZwierzakDomowy*, MUSZĄ te metody implementować (czyli przestępować).

Aby ZDEFINIOWAĆ interfejs:

```
public interface ZwierzakDomowy {...}
```

Zamiast słowa kluczowego „class” należy użyć słowa „interface”.

Aby ZIMPLEMENTOWAĆ interfejs:

```
public class Pies extends Psowate implements ZwierzakDomowy {...}
```

Zapisz słowo kluczowe „implements”, a po nim podaj nazwę interfejsu. Zauważ, że implementując interfejs, wciąż możesz rozszerzać inną klasę.

Tworzenie i implementacja interfejsu

ZwierzakDomowy

Używasz słowa kluczowego „interface” zamiast „class”.

```
public interface ZwierzakDomowy {
    public abstract void badzMilutki();
    public abstract void bawSie();
}
```

Niejawne metody interfejsów są publiczne i abstrakcyjne, a zatem wpisywanie słów kluczowych „public” i „abstract” nie jest konieczne (w rzeczywistości podawanie tych słów kluczowych jest jednak w tym przykładzie podaliśmy je, aby jawnie określić charakter metod oraz dlatego, że nigdy nie byliśmy niewolnikami mody...).

Pies JEST klasą Zwierze,
a jednocześnie Pies to
JEST ZwierzakDomowy

```
public class Pies extends Psowate implements ZwierzakDomowy {
    public void badzMilutki() {...}
    public void bawSie() {...}

    public void wedruj() {...}
    public void jedz() {...}
}
```

Wszystkie metody interfejsu są abstrakcyjne, zatem MUSZA się kończyć średnikiem. Pamiętaj, że takie metody nie mają żadnej treści.

Zapisujesz słowo „implements”, a po nim nazwę interfejsu.

POWIEDZIAŁEŚ, że tak klasa jest zwierzakiem domowym, zatem MUSISZ zaimplementować metody tworzące interfejs ZwierzakDomowy. To jest kontrakt. Zwróć uwagę na nawiasy klamrowe, które w tym przypadku zastąpiły znaki średnika.

A to są zwyczajne przestaniane metody.

Nieistniejąca grupa pytań

P: Chwileczkę, tak naprawdę to interfejsy nie zapewniają wielokrotnego dziedziczenia, gdyż nie można w nich umieszczać kodu implementującego metody. Jeśli wszystkie metody są abstrakcyjne, to w zasadzie co nam dają interfejsy?

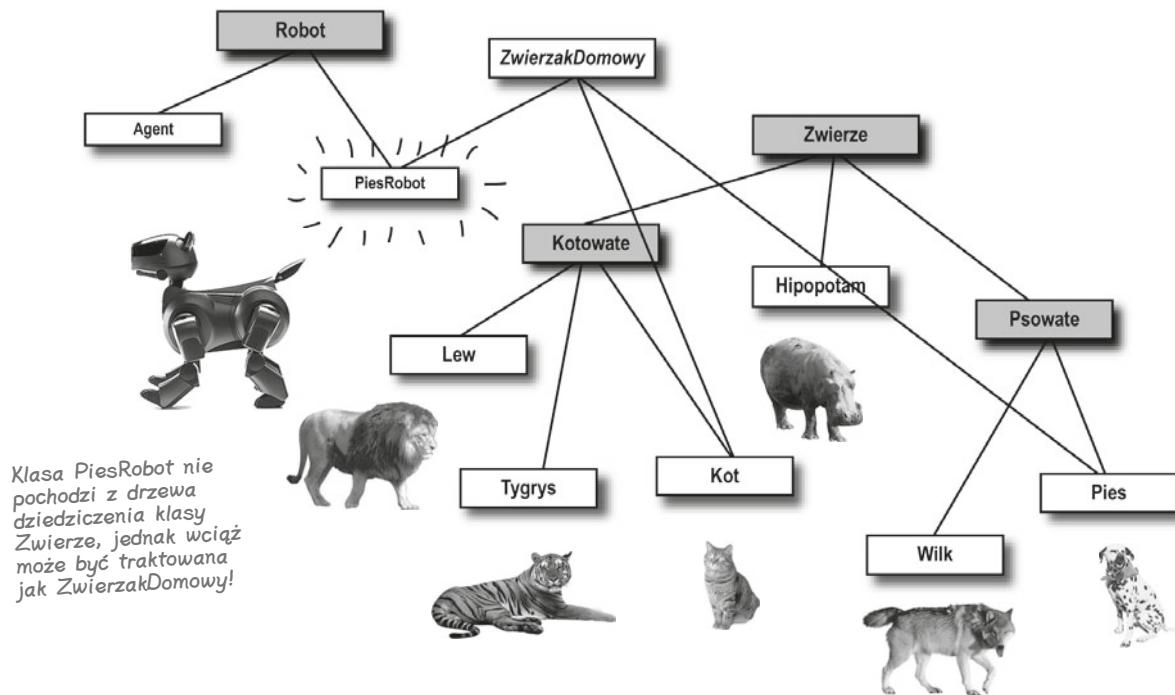
O: Polimorfizm, polimorfizm i jeszcze raz polimorfizm. Interfejsy są niezastąpione, jeśli chodzi o zapewnianie elastyczności. Jeśli bowiem zastosujesz interfejs zamiast konkretnych klas potomnych (lub nawet zamiast abstrakcyjnych klas bazowych) w argumentach metod lub wartościach

wynikowych, będziesz mógł przekazywać wszystko, co implementuje ten interfejs. Zastanów się — w przypadku zastosowania interfejsów, klasa nie musi pochodzić z jednego drzewa dziedziczenia. Klasa może rozszerzać inną klasę i zaimplementować jakiś interfejs. Jednak ten sam interfejs może zaimplementować także inna klasa, pochodząca z zupełnie innego drzewa dziedziczenia! A zatem możemy traktować obiekt, opierając się na funkcji, jaką pełni, a nie jego klasie. W rzeczywistości, gdybyś tworzył klasę z wykorzystaniem interfejsów, to nawet nie musiałbyś udostępniać klasy bazowej, którą one rozszerzają. Wystarczyłoby udostępnić

interfejs i ogłosić: „Hej! Nieważne, z jakiego rodzaju hierarchii klas pochodzicie, wystarczy jednak, że zaimplementujecie ten interfejs i możemy przystępować do dzieła”.

Co więcej, w przypadku dobrze zaprojektowanych klas, fakt, że w interfejsie nie można umieścić kodu implementującego metody, także nie jest wielkim problemem. Dzieje się tak dlatego, iż większość metod interfejsu i tak nie miałaby sensu, gdyby zostały zaimplementowane w sposób ogólny. Innymi słowy, i tak trzeba by przesłonić większość z metod interfejsu, nawet gdyby nie miały być metodami abstrakcyjnymi.

Ten sam interfejs mogą implementować klasy pochodzące z różnych drzew dziedziczenia.



Kiedy polimorficznym typem jest *klasa* (jak w przypadku tablicy typu Zwierze lub argumentu typu Psowate), obiekty, które mogą zostać uznane za obiekty tego typu, muszą należeć do tego samego drzewa dziedziczenia. Jednak nie mogą pochodzić z dowolnego miejsca tego drzewa — muszą to być obiekty klasy potomnej używanego typu polimorficznego. Argument typu Psowate może zaakceptować obiekty Pies lub Wilk, lecz nie obiekty Hipopotam lub Kot.

Jednak w sytuacji, gdy typem polimorficznym jest *interfejs* (jak w przypadku tablicy typu ZwierzakDomowy), obiekty mogą pochodzić z *dowolnego* miejsca drzewa dziedziczenia. Trzeba spełnić tylko jeden warunek — klasa obiektów musi *implementować* dany interfejs. Możliwość implementacji wspólnego interfejsu przez klasy należące do różnych hierarchii dziedziczenia ma kluczowe znaczenie dla Java API. Czy chcesz, aby obiekt był w stanie zapisać swój stan w pliku? Wystarczy że zaimplementujesz interfejs Serializable. Czy metody Twojego obiektu muszą być wykonywane w osobnym wątku? Zaimplementuj interfejs

Runnable. Pewnie już rozumiesz, o co chodzi. W dalszych rozdziałach znajdziesz więcej informacji o interfejsach Serializable oraz Runnable, jak na razie zapamiętaj, że mogą one być niezbędne w klasach zajmujących *dowolne* miejsca w hierarchii dziedziczenia. Możliwości zapisywania danych i uruchamiania w osobnych wątkach mogą być pożądane niemal *we wszystkich* klasach.

Ale to nie wszystko... klasa może implementować więcej niż jeden interfejs!

Obiekt Pies JEST obiektem klasy Psowate, jak również obiektem klasy Zwierze a także klasy Object, a wszystko to dzięki mechanizmowi dziedziczenia. Jednak Pies jest typu ZwierzakDomowy poprzez fakt zaimplementowania interfejsu, a klasa Pies może implementować także i inne interfejsy. Można by „rzuć”:

```
public class Pies extends Zwierze implements
ZwierzakDomowy,
Zapisywalny, UmożliwiającyWyswietlenie { ... }
```



Jak określić, czy stworzyć klasę, klasę potomną, klasę abstrakcyjną czy może interfejs?

- Utwórz klasę, która nie rozszerza żadnej innej (oczywiście za wyjątkiem klasy `Object`), jeśli nie sprawdza ona testu relacji JEST z żadną inną klasą.
- Utwórz klasę potomną (innymi słowy, rozszerz klasę) jedynie w przypadku, gdy potrzebujesz bardziej wyspecjalizowanej wersji klasy i musisz przesłonić jakieś zachowania lub dodać nowe.
- Użyj klasy abstrakcyjnej, jeśli chcesz zdefiniować pewien wzorzec dla grupy klas potomnych i jeśli dysponujesz choćby niewielkim fragmentem wspólnego kodu, z którego będą mogły korzystać wszystkie klasy potomne. Stwórz klasę abstrakcyjną, jeśli chcesz mieć pewność, że nikt nie będzie mógł tworzyć obiektów tego typu.
- Utwórz interfejs, jeśli chcesz zdefiniować funkcję, jaką mogą pełnić inne klasy i to niezależnie od ich położenia w drzewie dziedziczenia.

Wywoływanie wersji metody zdefiniowanej w klasie bazowej

P: A co zrobić w przypadku, gdy tworzę konkretną klasę potomną i muszę przesłonić w niej metodę klasy bazowej, jednak chciałbym mieć dostęp do tej metody klasy bazowej? Innymi słowy, co zrobić, jeśli nie muszę zastępować metody poprzez jej przesłonięcie, lecz chciałbym ją rozbudować, dodając do niej pewien kod?

O: Ehh... Zastanów się, co oznacza słowo „rozszerzać”. Zasady dobrego projektowania zorientowanego obiektowo starają się odpowiedzieć, między innymi, na pytanie, jak projektować solidny kod, który jest przeznaczony do tego, by go przesłonić. Innymi słowy, na przykład, w klasie abstrakcyjnej tworzysz kod metody, który wykonuje na tyle ogólne czynności, iż może być z powodzeniem stosowany w większości konkretnych implementacji. Jednak kod ten nie jest w stanie wykonać *wszystkich* wyspecjalizowanych zadań, jakie ma realizować klasa potomna. Zatem klasa potomna przeszła ogólną metodę i rozszerza ją, dodając resztę niezbędnego kodu. Do wywołania przesłoniętej metody klasy bazowej z poziomu klasy potomnej służy słowo kluczowe super.

Jeśli w klasie RaportPlotkarski pojawi się kod:

super.generujRaport();

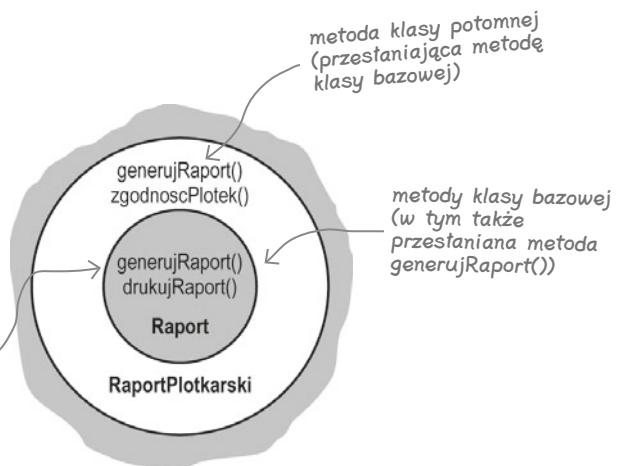
to zostanie wykonana wersja metody zdefiniowana w klasie bazowej.

super.generujRaport();

Użycie odwołania do obiektu klasy potomnej (RaportPlotkarski) zawsze spowoduje wywołanie metody klasy potomnej przestanającej metodę klasy bazowej. Tak działa polimorfizm. Jednak kod klasy potomnej może wykonać metodę klasy bazowej, używając w tym celu wywołania `super.generujRaport()`.

```
abstract class Raport {
    void generujRaport() { // tworzy raport
    }
    void drukujRaport() { // ogólny sposób drukowania raportów
    }
}
```

```
class RaportPlotkarski extends Raport {
    void generujRaport() {
        super.generujRaport(); // Wywołanie metody klasy bazowej oraz realizacja dodatkowych czynności.
        zgodnoscPlotek();
        drukujRaport();
    }
    void zgodnoscPlotek() { ... }
}
```



W rzeczywistości słowo kluczowe `super` jest odwołaniem do części obiektu stanowiącej obiekt klasy bazowej. Kiedy kod klasy potomnej używa tego słowa kluczowego, jak na przykład w wywołaniu `super.generujRaport()`, to wywoływana jest wersja metody zdefiniowana w klasie bazowej.



CELNE SPOSTRZEŻENIA

- Jeśli nie chcesz pozwolić na tworzenie obiektów pewnej klasy, oznacz ją przy użyciu słowa kluczowego **abstract**.
- W klasie abstrakcyjnej można umieszczać zarówno abstrakcyjne, jak i normalne metody.
- Jeśli klasa ma choćby *jedną* metodę abstrakcyjną, musi zostać oznaczona jako klasa abstrakcyjna.
- Metoda abstrakcyjna nie ma żadnej treści, a jej deklaracja kończy się średnikiem (nie są w niej zapisywane nawiasy klamrowe).
- Wszystkie metody abstrakcyjne muszą zostać zaimplementowane w pierwszej konkretnej klasie wchodzącej w skład drzewa dziedziczenia.
- Każda klasa w Javie jest jawną bądź niejawną klasą potomną klasy **Object** (`java.lang.Object`).
- W deklaracjach metod można używać zarówno argumentów typu **Object**, jak i wartości wynikowych tego typu.
- Metody obiektu można wywoływać *wyłącznie* w przypadku, gdy są one dostępne w klasie (lub interfejsie) będącej klasą zmiennej *referencyjnej*, niezależnie od faktycznej klasy danego obiektu. A zatem zmiennej referencyjnej typu **Object** można używać wyłącznie od wywoływania metod klasy **Object**, niezależnie od klasy obiektu, do jakiego zmienna ta się odwołuje.
- Zmienne referencyjne typu **Object** można przypisywać zmiennym referencyjnym jakiegokolwiek innego typu jedynie w przypadku użycia *rzutowania*. Rzutowanie można wykorzystać, by przypisać wartość zmiennej referencyjnej innej zmiennej referencyjnej, przy czym musi to być zmienna klasy potomnej. W czasie wykonywania programu rzutowanie się nie powiedzie, jeśli faktyczny obiekt znajdujący się na stercie NIE będzie zgodny z typem zastosowanym w rzutowaniu.
Przykład: **Pies d = (Pies) x.getObject(mojPies);**
- Wszystkie obiekty odczytywane z tablicy `ArrayList<Object>` są typu **Object** (co oznacza, że można się do nich odwoływać wyłącznie przy użyciu zmiennych referencyjnych klasy **Object**).
- W Javie wielokrotne dziedziczenie nie jest dozwolone, ze względu na potencjalne komplikacje wiążące się z problemem „śmiertelnego rombu”. Oznacza to, że można rozszerzać tylko jedną klasę (czyli, że klasa może mieć tylko jedną bezpośrednią klasę bazową).
- Interfejs można porównać z całkowicie abstrakcyjną klasą. Można w nim definiować wyłącznie metody abstrakcyjne.
- Tworząc interfejsy, zamiast słowa kluczowego **class** należy używać słowa kluczowego **interface**.
- Implementując interfejs, należy posłużyć się słowem kluczowym **implements**, na przykład **Pies implements ZwierzakDomowy**.
- Klasa może implementować więcej niż jeden interfejs.
- W klasie implementującej interfejs *trzeba* zaimplementować wszystkie jego metody, gdyż *w niejawnym sposobie wszystkie metody interfejsu są publiczne i abstrakcyjne*.
- Jeśli metoda klasy nadzędnej została prześloniona, to można ją wywołać posługując się słowem kluczowym **super**. Przykład: `super.generujRaport()`.

P: Wciąż widzę tu coś dziwnego... wciąż nie wyjaśniliście, dlaczego tablica `ArrayList<Pies>` zwraca odwołania do obiektów klasy **Pies**, których nie trzeba rzutować, a jednak klasa `ArrayList` wykorzystuje w swoich metodach obiekty **Object**, a nie **Pies** (ani **Portal**, ani innego typu). Co to za sztuczka z tym zapisem `ArrayList<Pies>`?

O: Masz rację, nazywając to specjalną sztuczką. To, że obiekt `ArrayList<Pies>` zwraca obiekty klasy **Pies** bez konieczności stosowania rzutowania, jest sztuczką, gdyż wygląda na to, że metody klasy `ArrayList` nie wiedzą o klasie **Pies** ani o jakimkolwiek innym typie oprócz **Object**.

Najkrótsza z możliwych odpowiedzi jest taka, że *to kompilator umieszcza w kodzie rzutowanie za Ciebie*. Pisząc `ArrayList<Pies>`, nie używamy żadnej specjalnej klasy, której metody mogą operować na obiektach klasy **Pies**, jednak zapis `<Pies>` jest dla kompilatora sygnałem, że chcemy, by w tablicy były zapisywane WYŁĄCZNIE obiekty **Pies**, i aby powstrzymał nas, jeśli spróbujemy zapisać w niej cokolwiek innego. A ponieważ kompilator pozwoli umieścić w tablicy tylko obiekty **Pies**, zatem wie on, że wszystko, co jest z niej pobierane, można bez najmniejszego ryzyka rzutować do obiektu klasy **Pies**. Innymi słowy, zastosowanie wyrażenia `ArrayList<Pies>` chroni nas przed koniecznością rzutowania obiektów pobieranych z tablicy do typu **Pies**. Jednak w rzeczywistości wyrażenie to ma znacznie ważniejsze konsekwencje... pamiętaj bowiem, że rzutowanie może zawieść w czasie działania programu; a raczej chciałbyś, żeby błędy pojawiły się w czasie komplikacji kodu, a nie w trakcie wykonywania programu o krytycznym znaczeniu dla Twojego klienta, prawda? Jednak na ten temat można powiedzieć znacznie więcej, co też uczyliśmy w rozdziale poświęconym kolekcjom.

Jaki będzie diagram?



Ćwiczenie

Oto masz okazję do zaprezentowania swoich uzdolnień artystycznych. Z lewej strony podane zostały grupy deklaracji klas i interfejsów. Twoim zadaniem jest narysowanie z prawej strony odpowiadających im diagramów klas. Pierwszy punkt ćwiczenia wykonaliśmy za Ciebie (i to za darmo!).

Dane

1) public interface Przesuwalny { }
public class Pasek implements Przesuwalny { }

2) public interface Zig { }
public abstract class Zag implements Zig { }

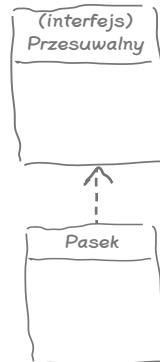
3) public abstract class Flip implements Klap { }
public class Flap extends Flip { }
public interface Klap { }

4) public class Kizi { }
public class Mizi extends Kizi { }
public class Klap extends Mizi { }

5) public class Gamma extends Delta implements Epsilon { }
public interface Epsilon { }
public interface Beta { }
public class Alfa extends Gamma implements Beta { }
public class Delta { }

Jaki będzie diagram?

1)



2)

3)

4)

5)

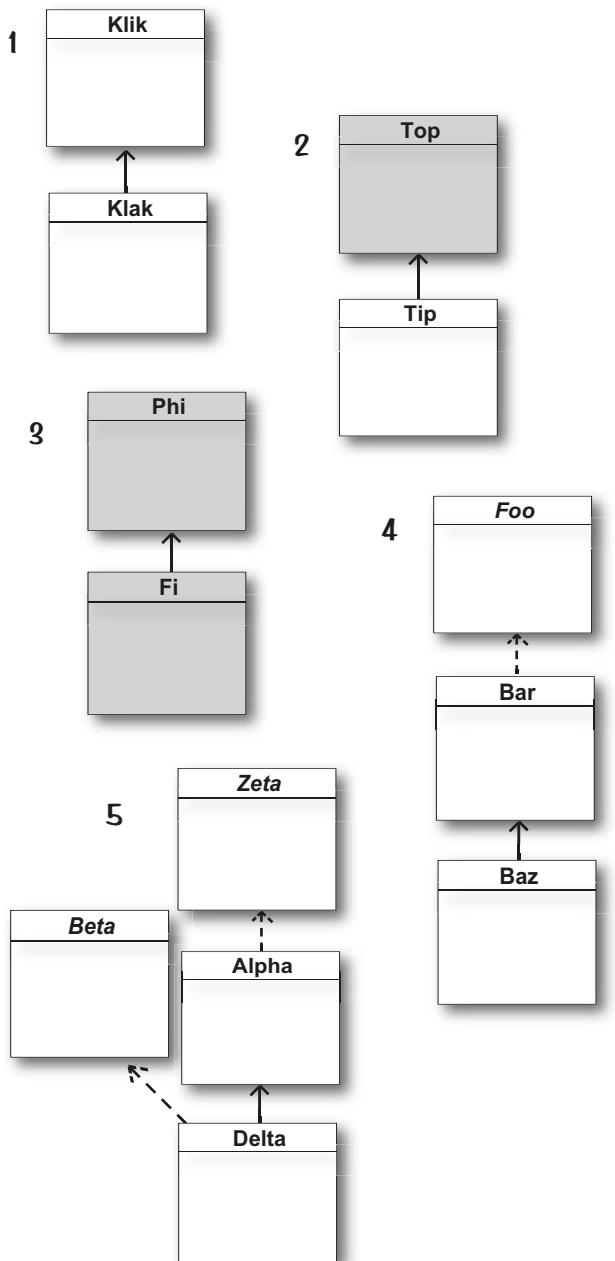


Ćwiczenie

Z lewej strony przedstawionych zostało kilka diagramów klas. Twoim zadaniem jest zamienienie ich na poprawne deklaracje klas i interfejsów. Za Ciebie wykonaliśmy już pierwszy punkt ćwiczenia (i to naprawdę trudny).

Jaka będzie deklaracja?

Diagram:



1) `public class Klik { }`
`public class Klak extends Klik { }`

2)

3)

4)

5)

Legenda

roszcza

implementuje

klasa

interfejs

klasa abstrakcyjna

Zagadka. Zagadkowy basen



Zagadkowy basen



Twoim **zadaniem** jest wybranie fragmentów kodu z basenu i umieszczenie ich w miejscach kodu oznaczonych podkreśleniami. Ten sam fragment kodu **może** być użyty więcej niż jeden raz, jednak może się zdarzyć, że nie wszystkie fragmenty zostaną wykorzystane. **Zadanie** polega na stworzeniu grupy klas, które będzie można skompilować i wykonać, i które wygenerują wyniki przedstawione poniżej.

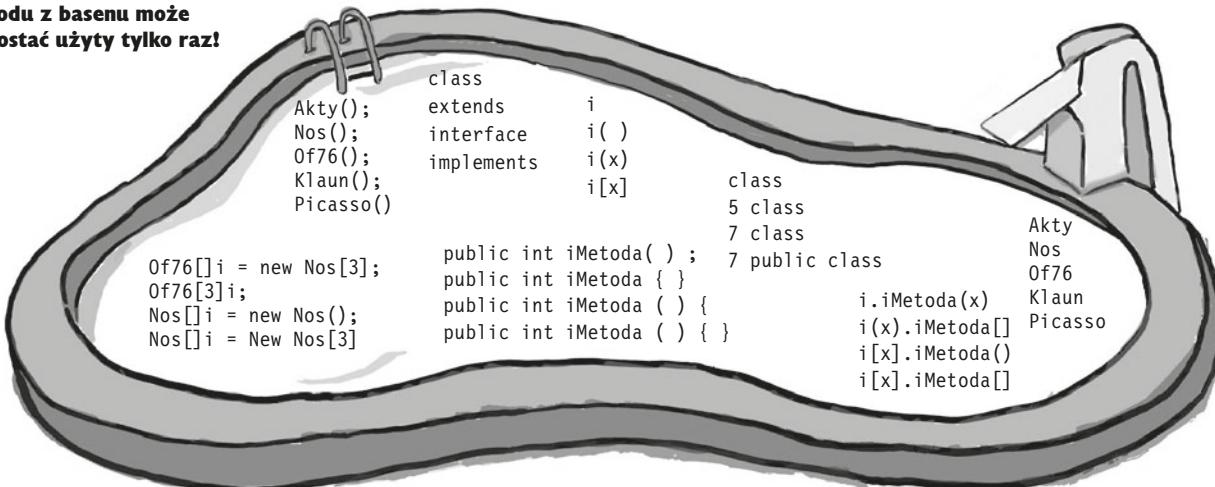
```
_____ Nos {  
_____ }  
  
abstract class Picasso implements _____ {  
_____ return 7;  
}  
}  
  
class _____ { }  
  
class _____ {  
_____ return 5;  
}  
}
```

Notatka: Każdy fragment kodu z basenu może zostać użyty tylko raz!

```
public _____ extends Klaun {  
public static void main(String[] args) {  
  
i[0] = new _____  
i[1] = new _____  
i[2] = new _____  
for (int x = 0; x < 3; x++) {  
System.out.println( _____  
+ " " + _____.getClass( ) );  
}  
}  
}
```

Wyniki:

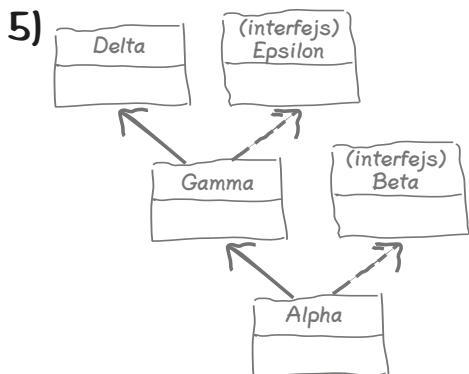
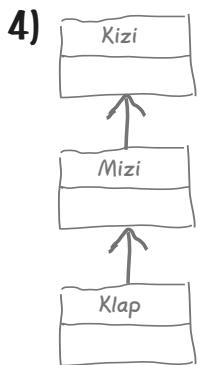
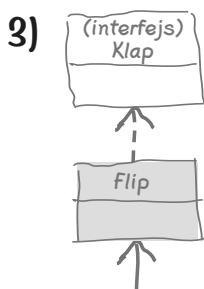
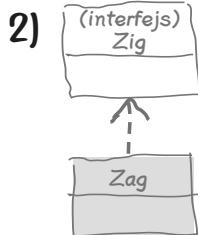
```
T:\>java _____  
5 class Akty  
7 class Klaun  
_____ Of76
```





Rozwiązania zadań

Jaki będzie diagram?



Jaka jest deklaracja?

2)

```
public abstract class Top { }
```

public class Tip extends Top { }

3)

```
public abstract class Phi { }
```

public abstract class Fi extends Phi { }

4)

```
public interface Foo { }
```

public class Bar implements Foo { }

public class Baz extends Bar { }

5)

```
public interface Zeta { }
```

public class Alfa implements Zeta { }

public interface Beta { }

public class Delta extends Alfa implements Beta { }

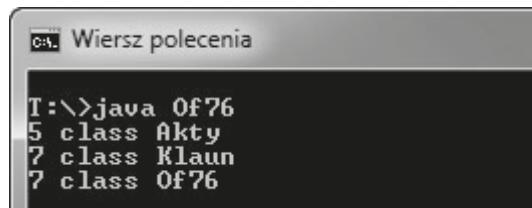
Rozwiążania zagadki



```
interface Nos {  
    public int iMetoda();  
}  
  
abstract class Picasso implements Nos  
{  
    public int iMetoda() {  
        return 7;  
    }  
}  
  
class Klaun extends Picasso {}  
  
class Akty extends Picasso {  
    public int iMetoda() {  
        return 5;  
    }  
}
```

```
public class Of76 extends Klaun {  
    public static void main(String[] args) {  
        Nos[] i = new Nos[3];  
        i[0] = new Akty();  
        i[1] = new Klaun();  
        i[2] = new Of76();  
        for (int x = 0; x < 3; x++) {  
            System.out.println( i[x].iMetoda()  
                + " " + i[x].getClass() );  
        }  
    }  
}
```

Wyniki:



The screenshot shows a terminal window titled "Wiersz polecenia" (Command Line). The command entered is "T:\>java Of76". The output displayed is:
5 class Akty
7 class Klaun
7 class Of76

9. Konstruktory i odśmiecacz

Życie i śmierć obiektu



...i wtedy powiedział. „Nie czuję nóg!”, a ja odpowiedziałem „Jasiu! Zostań ze mną, Jasiu!”. Ale... już było za późno. Pojawił się odśmiecacz i... było już po nim. Najlepszy obiekt, jaki kiedykolwiek miałem.

Obiekty się rodzą i obiekty umierają. To Ty odpowiadasz za ich cykl życia. Ty decydujesz, kiedy obiekt należy **utworzyć**. Również Ty decydujesz, kiedy obiekt należy **zniszczyć**. Z tym wyjątkiem, że tak naprawdę, to nie *niszczysz* obiektu, a jedynie go *porzucasz*. Jednak kiedy obiekt zostanie porzucony, bezduszny **odśmiecacz sterty** (czyli po prostu odśmiecacz) może go unicestwić, odzyskując tym samym pamięć, którą obiekt zajmował. Jeśli masz zamiar pisać programy w Javie, to będziesz tworzyć obiekty. Wcześniej czy później będziesz musiał któryś z nich usunąć lub ryzykować wyczerpanie pamięci operacyjnej. W tym rozdziale opiszemy, w jaki sposób obiekty są tworzone, gdzie są przechowywane podczas swojego istnienia oraz jak można je efektywnie zachować lub porzucić. Oznacza to, że będziemy pisać o stercie, stosie, zasięgu, konstruktorach, konstruktorach bazowych, odwołaniach pustych oraz wielu innych zagadnieniach. Ostrzeżenie: w rozdziale zostały podane informacje o umieraniu obiektów, które dla niektórych Czytelników mogą być szokujące. Zatem najlepszym rozwiązaniem będzie nienawiązywanie zbyt głębokich więzi emocjonalnych z obiektami.

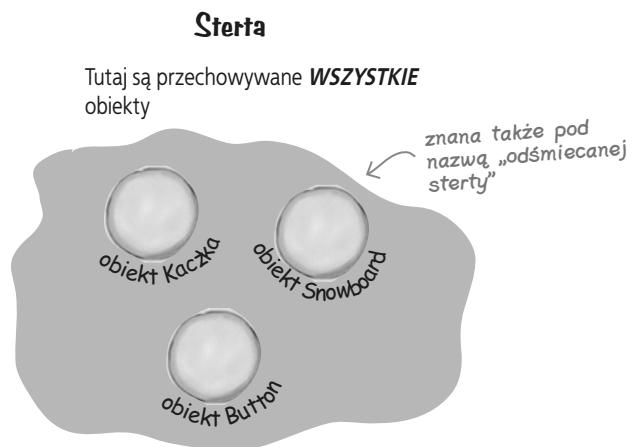
Stos i sterta. Gdzie są przechowywane informacje?

Zanim będziemy mogli zrozumieć, co się tak naprawdę dzieje podczas tworzenia obiektu, musimy się nieco cofnąć. Musimy się bowiem dowiedzieć, gdzie (i jak długo) „żyją” różne „rzeczy”. A to oznacza, że musimy dowiedzieć się czegoś więcej o stosie i stercie. Nas — programistów używających Javy — obchodzą dwa obszary pamięci — ten, w którym są umieszczane obiekty (czyli sterta), oraz ten, w którym umieszczone są wywołania metod oraz zmienne lokalne (czyli stos). Podczas uruchamiania wirtualnej maszyny Javy (JVM) otrzymuje ona od systemu operacyjnego fragment pamięci, który jest następnie używany do uruchamiania programów napisanych w Javie. To, ile pamięci zostanie przydzielonej oraz czy możesz ją w jakikolwiek sposób konfigurować, zależy od używanej wersji wirtualnej maszyny Javy (oraz systemu operacyjnego).



Jednak zazwyczaj *nie będziesz* miał nic do powiedzenia w tych sprawach, a jeśli Twoje programy będą tworzone poprawnie, to prawdopodobnie nie będzie Cię to obchodziło (więcej na ten temat dowieš się w dalszej części rozdziału).

Wiemy, że wszystkie *obiekty* są przechowywane na automatycznie odśmiecanej stercie, jednak nie wspominaliśmy o tym, gdzie są przechowywane *zmienne*. Nie wspominaliśmy także o tym, czy istnienie zmiennych zależy od ich *rodzaju*. Przy czym pisząc „rodzaj”, nie mamy na myśli *typu* (czyli jednego z typów podstawowych lub odwołania). Dwoma *rodzajami* zmiennych, jakie nas interesują, są *składowe* oraz zmienne *lokalne*. Te ostatnie są także nazywane zmiennymi stosowymi, co w oczywisty sposób określa miejsce ich przechowywania.



Składowe

Składowe są deklarowane wewnątrz *klasy*, lecz nie wewnątrz *metody*. Reprezentują one „pola”, jakie będzie posiadać każdy obiekt (i których wartości w każdym z tych obiektów mogą być inne). Składowe istnieją wewnątrz obiektu, do którego należą.

```
public class Kaczka {  
    int wielkosc;  
}  
Każdy obiekt Kaczka ma składową „wielkosc”.
```

Zmienne lokalne

Zmienne lokalne są deklarowane wewnątrz *metod*, przy czym zaliczamy do nich także parametry. Są to zmienne tymczasowe i istnieją do momentu, gdy metoda jest zapisana na stosie (innymi słowy, do chwili, gdy realizacja programu nie osiągnie zamkającego nawiasu klamrowego metody).

```
public void doRoboty(int x) {  
    int i = x + 3;  
    boolean b = true;  
}  
Parametr x oraz zmienne i oraz b są zmiennymi lokalnymi
```

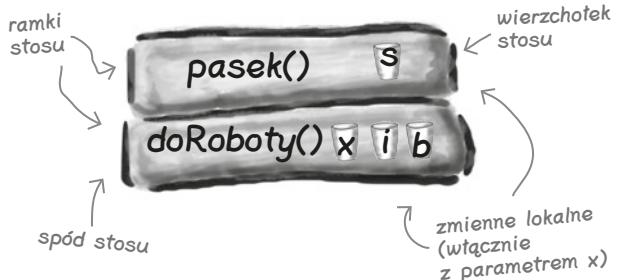
Metody są zapisywane na stosie

Kiedy wywołujesz metodę, jest ona umieszczana na wierzchołku stosu wywołań. Nowym elementem zapisywanym na stosie jest ramka stosu, która przechowuje informacje o stanie metody, w tym numer aktualnie wykonywanego wiersza kodu oraz wartości wszystkich zmiennych lokalnych.

Na wierzchołku stosu zawsze znajduje się metoda aktualnie wykonywana na danym stosie (jak na razie zakładamy, że istnieje tylko jeden stos, choć w rozdziale 14. dodamy ich więcej). Metoda pozostaje na stosie do chwili, gdy jej realizacja dotrze do zamykającego nawiasu klamrowego (co oznacza, że metoda została zakończona). Jeśli metoda `doRoboty()` wywoła metodę `pasek()`, to metoda `pasek()` zostanie zapisana na stosie powyżej `doRoboty()`.

```
public void doRoboty() {
    boolean b = true;
    jazda(4);
}
public void jazda(int x) {
    int z = x + 24;
    wariat();
    // wyobraź sobie, że to jest więcej kodu
}
public void wariat() {
    char c = 'a';
}
```

Stos wywołań, na którym zostały zapisane dwie metody.



Na samym wierzchołku stosu zawsze znajduje się aktualnie realizowana metoda.

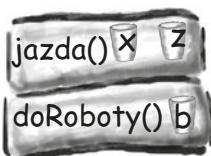
Przykład działania stosu

Z lewej strony przedstawiony został fragment kodu zawierający trzy metody (reszta kodu klasy nas nie interesuje). Pierwsza metoda (`doRoboty()`) wywołuje drugą (`jazda()`), która z kolei wywołuje trzecią (`wariat()`). Każda z metod deklaruje jedną zmienną lokalną, a metoda `jazda()` dodatkowo deklaruje jedną zmienną parametryczną (co oznacza, że metoda ta ma dwie zmienne lokalne).

- 1 Kod należący do innej klasy wywołuje metodę `doRoboty()`, ramka tej metody jest zapisywana na samym wierzchołku stosu. Zmienna lokalna o nazwie „`b`” jest umieszczana w ramce metody `doRoboty()`.



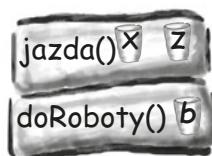
- 2 Metoda `doRoboty()` wywołuje metodę `jazda()`, która jest zapisywana na stosie. W ramce metody `jazda()` znajdują się zmienne `x` oraz `z`.



- 3 Metoda `jazda()` wywołuje metodę `wariat()`, teraz metoda `wariat()` zajmuje miejsce na wierzchołku stosu, a w jej ramce jest umieszczana zmienna `c`.



- 4 Wykonywanie metody `wariat()` zostaje zakończone, a jej ramka jest zdejmowana ze stosu. Realizacja programu trafia ponownie do metody `jazda()`, gdzie jest kontynuowana od pierwszego wiersza poniżej wywołania metody `wariat()`.



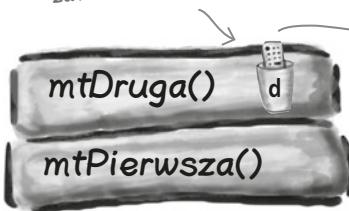
A co ze zmiennymi lokalnymi, które są obiektami?

Pamiętaj, że wszystkie zmienne, których typ nie jest jednym z typów podstawowych, zawierają odwołania do obiektów, a nie same obiekty. Przecież już wiesz, gdzie są przechowywane obiekty — na stercie. I nie ma najmniejszego znaczenia, gdzie zostały zadeklarowane lub utworzone. *Jeśli zmienna jest odwołaniem do obiektu, to jedynie sama zmienna (odwołanie — czyli „pilot”) jest przechowywana na stosie.*

Sam obiekt wciąż jest przechowywany na stercie.

```
public class OdwołanieDS {  
    public void mtPierwsza() {  
        mtDruga();  
    }  
    public void mtDruga() {  
        Kaczka d = new Kaczka(24);  
    }  
}
```

Metoda mtDruga() deklaruje i tworzy nową zmienną referencyjną „d” typu Kaczka (ponieważ została ona zadeklarowana wewnątrz metody, jest zatem zmienną lokalną i trafia na stos).



Niezależnie od tego, GDZIE została zadeklarowana zmienna referencyjna (w metodzie czy też wewnątrz klasy jako składowa), sam obiekt zawsze powtarzam zawsze, trafia na stertę.

Nie istnieją głupie pytania

P: Jeszcze raz, po co w ogóle uczymy się o tej stercie i stosie? Jak to może mi pomóc? Czy naprawdę muszę się tego uczyć?

O: Znajomość podstaw wykorzystania stosu i sterty w Javie ma kluczowe znaczenie dla zrozumienia zagadnień związanych z zasięgiem zmiennych, tworzeniem obiektów, zarządzaniem pamięcią, działaniem wątków oraz obsługą wyjątków. Wątki oraz obsługa wyjątków zostaną opisane w kolejnych rozdziałach, jednak pozostałe przedstawimy już teraz. Nie musisz znać sposobu implementacji stosu ani sterty w żadnej konkretnej wirtualnej maszynie Javy. Wszystko, co musisz wiedzieć na temat stosu i sterty, zostało podane na tej oraz na poprzedniej stronie. Jeśli uważnie je przeczytasz i przyswoisz sobie podane na nich informacje, to znacznie łatwiej będzie Ci zrozumieć inne zagadnienia, które ściśle się wiążą z wykorzystaniem stosu i sterty. Powtórzmy jeszcze raz — pewnego dnia będziesz nam bardzo wdzięczny, że wbiliśmy Ci do głowy trochę informacji o stosie i stercie.



CELNE SPOSTRZEŻENIA

- W Javie używane są dwa obszary pamięci, które nas interesują — stos oraz sterta.
- Składowe są zmiennymi deklarowanymi wewnątrz klas, lecz poza metodami.
- Zmienne lokalne są zmiennymi deklarowanymi wewnątrz metod lub jej parametrami.
- Wszystkie zmienne lokalne istnieją na stosie, w ramce odpowiadającej metodzie, w której zostały zadeklarowane.
- Zmienne przechowujące odwołania do obiektów działają tak samo jak zmienne typów podstawowych — jeśli odwołanie zostało zadeklarowane jako zmienna lokalna, to będzie przechowywane na stosie.
- Wszystkie obiekty są przechowywane na stercie, niezależnie od tego, czy odwołania wskazujące na nie są składowymi czy też zmiennymi lokalnymi.

Jeśli zmienne lokalne są przechowywane na stosie, to gdzie są przechowywane składowe?

Jeśli Java napotka instrukcję `new TelefonKomorkowy()`, będzie musiała zarezerwować na stercie obszar przeznaczony na obiekt `TelefonKomorkowy`. Ale jak duży będzie ten obszar? Wystarczający do przechowania obiektu czy na tyle duży, aby pomieścić wszystkie jego składowe. Wszystko w porządku, składowe są przechowywane na stercie, wewnątrz obiektów, do których należą.

Pamiętaj, że *wartości* składowych obiektu są przechowywane wewnątrz obiektów. Jeśli te składowe są typów podstawowych, to Java rezerwuje dla nich miejsce, którego wielkość jest określana na podstawie typów. Zmienna typu `int` potrzebuje 32 bity, zmienna typ `long` — 64 i tak dalej. Java nie zwraca uwagi na wartości przechowywane w zmiennych typów podstawowych; pod względem ilości bitów zmienna typu `int` jest taka sama, niezależnie od tego, czy ma wartość 32 000 000 czy też 32.

Co się jednak dzieje w sytuacji, gdy składowe są *obiektami*? Co jeśli `TelefonKomorkowy` MA składową `Antena`? Innymi słowy, co się dzieje, jeśli klasa `TelefonKomorkowy` zawiera zmienną referencyjną klasy `Antena`?

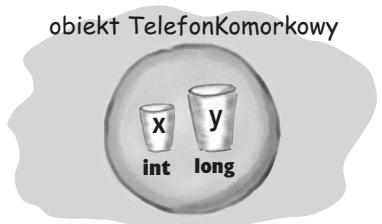
W przypadku gdy nowy obiekt zawiera składowe przechowujące odwołania, a nie wartości typów podstawowych, to prawdziwe pytanie, jakie należy postawić, to: „Czy obiekt potrzebuje miejsca na wszystkie inne obiekty, do których ma odwołania?”. Odpowiedź na nie brzmi: *Niezupelnie*. Niezależnie od wszystkiego Java rezerwuje miejsce na *wartości* składowych. Pamiętaj jednak, że wartościami zmiennych referencyjnych nie są same *obiekty*, lecz *odwołania*, czyli „piloty” tych obiektów. A zatem, jeśli w klasie `TelefonKomorkowy` została zadeklarowana składowa typu `Antena`, który nie jest jednym z typów podstawowych, to Java zarezerwuje w obiekcie `TelefonKomorkowy` miejsce na *pilota* (zmienną referencyjną), a nie na sam *obiekt* `Antena`.

Dobrze, zatem kiedy na stercie zostaje przydzielone miejsce na *obiekt* `Antena`? Przede wszystkim należy określić, *kiedy* ten obiekt jest tworzony. To z kolei zależy od deklaracji składowej. Jeśli zadeklarujemy składową, lecz nie zapiszemy w niej żadnego obiektu, to zarezerwowane zostanie jedynie miejsce na zmienną referencyjną (czyli na pilota do obiektu).

```
private Antena ant;
```

Obiekt `Antena` zostanie utworzony na stercie dopiero, gdy zmiennej referencyjnej przypiszymy nowy obiekt tej klasy:

```
private Antena ant = new Antena();
```

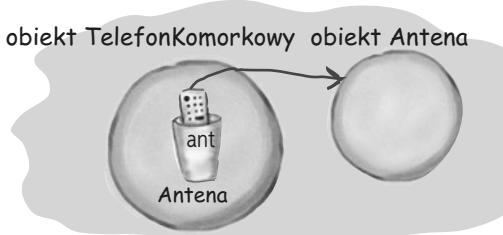


Obiekt ma dwie składowe typów podstawowych. Obszar na te zmienne znajduje się wewnątrz obiektu.



Obiekt, który nie ma składowych typów podstawowych. Jego składowa jest odwołaniem do obiektu `Antena`, lecz nie jest samym obiektem. Taki wynik można uzyskać, jeśli zadeklarujemy zmienną, lecz jej nie zainicjalizujemy, zapisując w niej faktyczny obiekt `Antena`.

```
public class TelefonKomorkowy {
    private Antena ant;
}
```



Obiekt ma składową będącą odwołaniem. W składowej został zapisany nowy obiekt `Antena`.

```
public class TelefonKomorkowy {
    private Antena ant = new Antena();
}
```

Cud utworzenia obiektu

Teraz, kiedy już wiesz, gdzie są przechowywane zmienne i obiekty, możemy już wkroczyć do tajemniczego świata tworzenia obiektu. Przypomnij sobie trzy etapy deklarowania i przypisywania obiektu. Są nimi: deklaracja zmiennej referencyjnej, utworzenie obiektu i przypisanie obiektu do zmiennej.

Aż do tej chwili drugi etap tego procesu, czyli moment, kiedy następuje cud i „rodzi” się nowy obiekt, pozostawał „wielką tajemnicą”. Przygotuj się na poznanie faktów dotyczących życia obiektu. *Miejmy nadzieję, że jesteś odporny!*

Przypomnijmy sobie trzy etapy deklarowania, tworzenia i przypisywania obiektu:

Utworzenie nowej zmiennej referencyjnej, której typ jest klasą lub interfejsem.

1 Deklaracja zmiennej referencyjnej

Kaczka mojaKaczka = new Kaczka();



I tu następuje cud.

2 Utworzenie obiektu

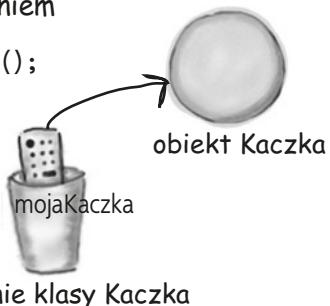
Kaczka mojaKaczka = new Kaczka() ;



Przypisanie nowego obiektu do odwołania.

3 Połączenie obiektu z odwołaniem

Kaczka mojaKaczka = new Kaczka() ;



Czy wywołujemy metodę o nazwie Kaczka()?

Kaczka mojaKaczka = new Kaczka();

Ze względu na nawiasy, wygląda to tak, jak gdyby wywoływana była metoda o nazwie Kaczka().

Nie.

Wywołujemy konstruktor klasy Kaczka.

Konstruktor *faktycznie* wygląda i działa bardzo podobnie do metody, jednak metodą nie jest. Zawiera kod, który zostaje wykonany w momencie użycia operatora **new**. Innymi słowy, *zawiera kod wykonywany w momencie tworzenia nowego obiektu*.

Jedynym sposobem wywołania konstruktora jest użycie słowa kluczowego **new** i podanie po nim nazwy klasy. W takim przypadku, wirtualna maszyna Javy odnajduje klasę i wywołuje jej konstruktor. (No dobrze, z technicznego punktu widzenia nie jest to *jedyny* sposób wywołania konstruktora. Jednak jest to *jedyny* sposób, w jaki można go wywołać *spoza* konstruktora, bowiem *można* wywołać konstruktor z poziomu innego konstruktora. Co prawda obowiązują przy tym pewne ograniczenia, ale do tego dojdziemy w dalszej części rozdziału).

Konstruktor zawiera kod, który jest wykonywany w momencie tworzenia obiektu. Innymi słowy, zawiera on kod wykonywany w przypadku użycia operatora **new** i nazwy danej klasy.

Każda tworzona klasa posiada konstruktor, nawet jeśli nie napiszesz go samemu.

Ale gdzie jest konstruktor?

Jeśli myśmy go nie stworzyli, to kto to zrobił?

Możesz sam napisać konstruktor dla swojej klasy (my mamy zamiar tak robić), jeśli jednak tego nie zrobisz, *to kompilator stworzy go za Ciebie!*

Poniżej przedstawiony został wygląd domyślnego konstruktora tworzonego przez kompilator:

```
public Kaczka() {  
}
```

Zauważłeś, że czegoś tu brakuje?

Czym powyższy kod różni się od metody?

Nazwa jest taka sama jak nazwa klasy. To konieczne.

```
public Kaczka() {  
    // tu zostaje umieszczony kod konstruktora  
}
```

A gdzie jest typ wyniku? Gdyby to była metoda, to pomiędzy „public” a „Kaczka” powinien zostać określony typ zwracanego wyniku.

Tworzenie obiektu Kaczka

Podstawową cechą konstruktora jest to, iż jest on wykonywany, *zanim* obiekt można skojarzyć z odwołaniem. Dzięki temu mamy możliwość wkroczenia do akcji i wykonania czynności, które przygotują obiekt do użycia. Innymi słowy, zanim ktokolwiek będzie miał okazję użyć pilota do obiektu, obiekt ma możliwość pomóc nam w jego tworzeniu. Nasz przykładowy konstruktor obiektu Kaczka nie robi niczego pozytycznego, jednak dobrze demonstruje sekwencję wydarzeń.

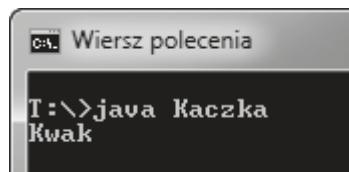
```
public class Kaczka {  
    public Kaczka() {  
        System.out.println("Kwak");  
    }  
}  
  
public class UzyjKaczki {  
    public static void main(String[] args) {  
        Kaczka k = new Kaczka();  
    }  
}
```

Kod konstruktora

Ten fragment kodu wywołuje konstruktor klasy Kaczka



Konstruktor daje możliwość ingerencji w proces tworzenia obiektu realizowany przez operator new.



Zaostrz ołówek

Konstruktor pozwala ingerować w proces tworzenia nowego obiektu — proces realizowany przez operator **new**. Czy potrafisz wyobrazić sobie okoliczności, w jakich taka możliwość mogłaby się przydać? Które z wymienionych czynności mogłyby być przydatne w konstruktorze klasy Samochód, zakładając, że jest ona wykorzystywana w klasie *Wyscigi Samochodów?* Zaznacz czynności, dla których możesz znaleźć zastosowanie.

- Inkrementacja licznika przechowującego informacje, ile obiektów tej klasy zostało utworzonych.
- Przypisanie stanu bieżącego (informacji o tym, co się dzieje w bieżącej chwili).
- Określenie wartości ważnych składowych obiektu.
- Pobranie i zapisanie odwołania do obiektu, który tworzy ten obiekt.
- Dodanie obiektu do tablicy `ArrayList`.
- Utworzenie obiektów będących składowymi tego obiektu (które spełniają relację MA).
- _____ (tu wpisz swoje pomysły).

Inicjalizacja stanu nowego obiektu Kaczka

Nie istnieja głupie pytania

W większości przypadków konstruktory używane są do inicjalizacji stanu obiektu. Innymi słowy — do określania wartości jego składowych.

```
public Kaczka() {
    wielkosc = 34;
}
```

Wszystko jest jasne i proste, o ile *programista* tworzący klasę Kaczka wie, jakie wielkości powinien być nowy obiekt tej klasy. Jednak co zrobić, jeśli chcemy, aby to programista *używający* obiektu Kaczka decydował, jaka ma być jego wielkość?

Wyobraźmy sobie, że klasa Kaczka ma składową określającą wielkość, oraz że chcemy, aby programista używający tej klasy określał wielkość każdego obiektu. W jaki sposób można to osiągnąć?

Cóż, można zdefiniować w klasie metodę zapisującą `setWielkosc()`. Jednak w takim przypadku przez pewien okres czasu, wielkość obiektu nie będzie określona*, a poza tym rozwiązanie to zmusza programistę do użycia dwóch instrukcji — pierwszej do stworzenia obiektu Kaczka i drugiej do wywołania metody `setWielkosc()`. Takie rozwiązanie zostało przedstawione na poniższym przykładzie:

```
public class Kaczka {
    int wiekosc;           ← składowa
    public Kaczka() {
        System.out.println("Kwak"); ← konstruktor
    }
    public void setWielkosc(int nowaWielkosc) {
        wiekosc = nowaWielkosc;   ← metoda ustawiająca
    }
}
```

```
public class UzyjKaczki {
    public static void main(String[] args) {
        Kaczka k = new Kaczka();           ← tutaj widać poważną wadę tego rozwiązania.
        k.setWielkosc(42);                ← W tym miejscu kodu obiekt Kaczka już istnieje,
                                         jednak nie została jeszcze określona jego
                                         wielkość! Poza tym, musimy polegać na tym,
                                         iż użytkownik klasy Kaczka WIE, że proces
                                         tworzenia obiektów tej klasy składa się z dwóch
                                         etapów — wywołania konstruktora
                                         oraz wywołania metody ustawiającej.
```

* Składowe mają wartości domyślne. W przypadku liczbowych typów podstawowych są to wartości 0 oraz 0.0, w przypadku składowych logicznych — false lub true, a dla odwołań — null.

P: Dlaczego trzeba pisać konstruktory, skoro kompilator może je tworzyć za nas?

O: Jeśli potrzebujesz kodu, który zainicjuje obiekt i przygotuje go do użycia, to będziesz musiał stworzyć konstruktor. Na przykład w celu zakończenia przygotowywania obiektu do użycia może być konieczne pobranie danych od użytkownika. Istnieje jeszcze jeden powód, dla którego możesz być zmuszony do stworzenia konstruktora, nawet jeśli sam kod umieszczany w konstruktorze nie jest konieczny. Powód ten jest związany z konstruktorami klas bazowych i przedstawimy go w dalszej części rozdziału.

P: Jak można odróżnić konstruktor od metody? Czy mogą także istnieć metody, których nazwy są takie same jak nazwa klasy?

O: Java daje możliwość deklarowania metod o takich samych nazwach jak nazwa klasy. Posiadanie tej samej nazwy nie czyni z nich konstruktorów. Czynnikiem, który odróżnia metodę od konstruktora, jest typ wartości wynikowej. Metody *muszą* zwracać jakąś wartość wynikową, natomiast konstruktory *nie mogą*.

P: Czy konstruktory są dziedziczone? Jeśli nie stworzę konstruktora, lecz będę on dostępny w klasie bazowej, to czy zamiast konstruktora domyślnego zostanie odziedziczony konstruktor po klasie bazowej?

O: Nie — konstruktory nie są dziedziczone. W dalszej części rozdziału podamy więcej informacji na ten temat.

Zastosowanie konstruktora do inicjalizacji ważnych aspektów stanu obiektu Kaczka*

Jeśli obiekt Kaczka nie powinien być używany w przypadku, gdy jedna lub więcej składowych określających jego stan nie zostanie zainicjalizowanych, to nie należy dawać nikomu dostępu do tego obiektu aż do momentu zakończenia jego inicjalizacji! Zazwyczaj umożliwienie komuś utworzenia nowego obiektu Kaczka (i pobranie odwołania do niego), który nie będzie w pełni gotowy do użycia, dopóki ktoś nie wywoła metody `setWielkosc()`, jest zbyt ryzykowne. Skąd użytkownik klasy Kaczka w ogóle będzie *wiedzieć*, że musi wywołać odpowiednią metodę ustawiającą po utworzeniu obiektu?

Najlepszym miejscem do umieszczenia kodu inicjalizującego jest konstruktor. A cały problem sprowadza się do utworzenia konstruktora z argumentami.

```
public class Kaczka {  
    int wielkosc;  
  
    public Kaczka(int wielkosCK) {  
        System.out.println("Kwak");  
        wielkosc = wielkosCK;  
        System.out.println("Wielkość kaczki: " + wielkosc);  
    }  
}
```

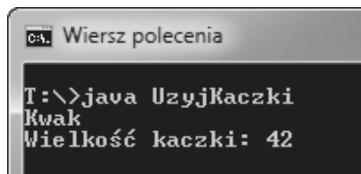
Do konstruktora klasy Kaczka dodajemy parametr typu int.

Używamy wartości argumentu, aby określić wartość składowej.

```
public class UzyjKaczki {  
    public static void main(String[] args) {  
        Kaczka k = new Kaczka(42);  
    }  
}
```

Tym razem używana jest tylko jedna instrukcja — tworzy ona nowy obiekt klasy Kaczka i jednocześnie określa jego wielkość.

Przekazujemy wartość do konstruktora.



* Przy czym nie sugerujemy tutaj, że istnieją jakieś aspekty stanu obiektu Kaczka, które są mniej ważne.

Pozwólmy użytkownikom jednocześnie utworzyć obiekt Kaczka i określić jego wielkość, wykorzystując przy tym operator new i wywołując konstruktor klasy Kaczka.



Ułatw tworzenie obiektów klasy Kaczka

Upewnij się, że stworzyłeś także konstruktor bezargumentowy

Co się dzieje w przypadku, gdy konstruktor klasy Kaczka wymaga podania argumentu? Przemyśl to. W przykładzie przedstawionym na poprzedniej stronie stworzony został *tylko jeden* konstruktor — konstruktor, który wymaga przekazania liczby typu `int` określającej *wielkość kaczki*. Być może nie jest to żaden poważny problem, niemniej jednak utrudnia programistom tworzenie nowych obiektów Kaczka, zwłaszcza jeśli *nie wiedzą*, jaka powinna być ich wielkość. Zatem może przydatne byłoby określenie domyślnej wielkości nowych kaczek? W ten sposób, nawet gdyby użytkownik nie znał właściwej wielkości, i tak mógłby utworzyć obiekt nadający się do wykorzystania.

Załóżmy, że chciałbys, aby użytkownicy klasy Kaczka mieli dwie możliwości. Pierwszą byłoby podanie wielkości tworzonego obiektu Kaczka (jako argumentu konstruktora), a drugą — pominiecie określenia wielkości i utworzenie obiektu o wielkości domyślnej.

Takiego rozwiązania nie można stworzyć w elegancki sposób, posługując się tylko jednym konstruktorem. Pamiętaj, że jeżeli w metodzie (albo konstruktorze, gdyż w tym przypadku obowiązują te same zasady) został zadeklarowany parametr, to w jej wywołaniu *trzeba* przekazać argument. Nie można tak po prostu powiedzieć: „Jeśli ktoś nie przekaże żadnego argumentu do konstruktora, to należy użyć wielkości domyślnej”. Takiego kodu nie dałoby się nawet skompilować bez przekazania, w wywołaniu konstruktora, odpowiedniego argumentu. Można by zastosować nieeleganckie rozwiązania przedstawione poniżej:

```
public class Kaczka {
    int wielosc;

    public Kaczka(int wieloscK) {
        if (wieloscK == 0) {
            wielosc = 27;
        } else {
            wielosc = wieloscK;
        }
    }
}
```

Jeśli wartość parametru jest równa 0, to składowej `wielosc` nadajemy wartość domyślną, w przeciwnym razie składowej przypisujemy wartość parametru. To NIE jest szczególnie dobre rozwiązanie.

Zastosowanie powyższego rozwiązania oznacza, że programista tworzący nowy obiekt `Kaczka` musi *wiedzieć*, iż przekazanie w wywołaniu konstruktora wartości 0 jest protokołem pozwalającym nadać nowemu obiekowi domyślną wielkość. A co się stanie, jeśli programista nie będzie o tym wiedzieć? Lub jeśli będzie chciał utworzyć obiekt o zerowej wielkości? (Oczywiście zakładając, że tworzenie obiektów `Kaczka` o zerowej wielkości byłoby dopuszczalne. Gdybyś chciał uniemożliwić tworzenie takich obiektów, wystarczyłoby dodać do konstruktora odpowiedni kod sprawdzający). Najważniejsze jest to, iż nie zawsze byłoby możliwe rozróżnienie, kiedy programista naprawdę chce uzyskać obiekt `Kaczka` o zerowej wielkości, a kiedy jedynie stwierdza: „Przekazuję zero, zatem utwórz obiekt o domyślnej wielkości, jakakolwiek by ona nie była”.

Tak naprawdę chcesz mieć DWA sposoby tworzenia nowych obiektów Kaczka:

```
public class Kaczka2 {
    int wielosc;

    public Kaczka2() {
        // określenie domyślnej
        // wielkości
        wielosc = 27;
    }

    public Kaczka2(int wieloscK) {
        // użycie parametru
        wielosc = wieloscK;
    }
}
```

Aby utworzyć nowy obiekt znając jego wielkość, użyj instrukcji:

```
Kaczka2 k = new Kaczka2(15);
```

Aby utworzyć nowy obiekt, jeśli nie będziesz znać jego wielkości, użyj instrukcji:

```
Kaczka2 k = new Kaczka2();
```

A zatem pomysł tworzenia nowych obiektów `Kaczka` na dwa sposoby wymaga zastosowania dwóch konstruktorów. Jednego, który wymaga przekazania liczby całkowitej, oraz drugiego — bezargumentowego.

Jeśli w klasie istnieje więcej niż jeden konstruktor, oznacza to, że konstruktory zostały przeciążone.

Czy kompilator zawsze tworzy dla nas konstruktor bezargumentowy?

Nie!

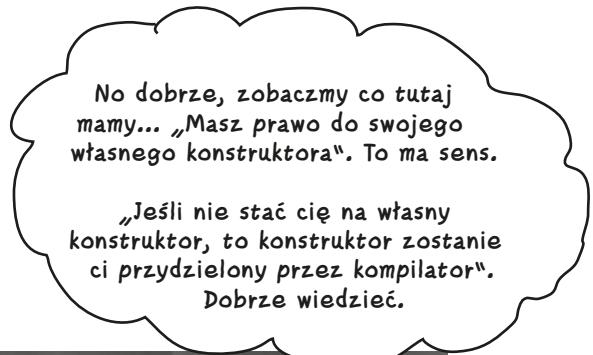
Mogłybyś przypuszczać, że gdy stworzysz wyłącznie konstruktor wymagający przekazania argumentów, to kompilator to zauważ i samodzielnie doda konstruktor bezargumentowy. Jednak tak się nie dzieje. Kompilator zajmuje się tworzeniem konstruktorów *wyłącznie wtedy, jeśli Ty w ogóle się na ich temat nie wypowiesz.*

Jeśli stworzysz konstruktor pobierający argumenty, lecz pomimo to wciąż chcesz dysponować także konstruktorem bezargumentowym, to będziesz go musiał stworzyć sam!

Jeśli stworzysz jakikolwiek konstruktor, kompilator wycofa się i stwierdzi: „Dobra, stary, wygląda na to, że to Ty chcesz zarządzać konstruktorami”.

Jeśli w klasie jest więcej niż jeden konstruktor, to każdy z nich **MUSI mieć inną listę argumentów.**

Na listę argumentów składa się kolejność argumentów oraz ich typy. W klasie można zdefiniować więcej niż jeden konstruktor, o ile ich listy argumentów będą różne. To samo dotyczy metod, lecz tym zagadnieniem zajmiemy się w jednym z kolejnych rozdziałów.



Przeciążenie konstruktorów oznacza, że w klasie został zdefiniowany więcej niż jeden konstruktor.

Aby klasę zawierającą kilka konstruktorów można było skompilować, każdy z nich musi mieć inną listę argumentów!

Przedstawiona poniżej klasa jest poprawna, gdyż wszystkie pięć konstruktorów ma różne listy argumentów. Gdybyś stworzył dwa konstruktory pobierające, na przykład, wyłącznie liczbę typu int, to takiej klasy nie można by skompilować. Nazwy parametrów nie mają w tym przypadku żadnego znaczenia. Liczy się typ zmiennych (int, Pies itd.) oraz ich kolejność. Można stworzyć dwa konstruktory o takich samych typach parametrów, o ile kolejność tych parametrów będzie różna. Konstruktor wymagający przekazania łańcucha znaków i liczby typu int różni się od konstruktora wymagającego przekazania liczby typu int i łańcucha znaków.

Pięć różnych konstruktorów oznacza pięć różnych sposobów na utworzenie nowego grzybka.



Te dwa konstruktory mają te same typy argumentów, lecz argumenty są przekazywane w innej kolejności; zatem wszystko jest w porządku.

```
public class Grzybek { }           Jeżeli znasz wielkość grzybka,  
                                    ale nie wiesz, czy jest magiczny.  
  
public Grzybek(int wielkosc) {}   Jeżeli nic nie wiesz na temat grzybka.  
  
public Grzybek() {}               Jeżeli wiesz, czy grzybek jest magiczny  
                                    czy nie, ale nie znasz jego wielkości.  
  
public Grzybek(boolean czyMagiczny) {}  
public Grzybek(boolean czyMagiczny, int wielkosc) {} } Kiedy wiesz, czy  
public Grzybek(int wielkosc, boolean czyMagiczny) {} } grzybek jest magiczny  
}                                         czy nie oraz znasz jego wielkość.
```



CELNE SPOSTRZEŻENIA

- Składowe istnieją wewnętrz obiektu, do którego należą, na stercie.
- Jeśli składowa jest odwołaniem do obiektu, to zarówno odwołanie, jak i obiekt, na który ono wskazuje, są przechowywane na stercie.
- Konstruktor to kod wykonywany w przypadku użycia operatora **new** i podania nazwy klasy.
- Konstruktor musi mieć taką samą nazwę jak klasa i *nie* może zawierać określenia typu wartości wynikowej.
- Konstruktora można używać do inicjalizacji stanu (czyli składowych) tworzonego obiektu.
- Jeśli sam nie stworzysz w klasie żadnego konstruktora, to kompilator doda do niej konstruktor domyślny.
- Konstruktor domyślny zawsze jest konstruktorem bezargumentowym.
- Jeśli zdefiniujesz w klasie jakikolwiek konstruktor, to kompilator nie doda do niej konstruktora domyślnego.
- Jeśli chcesz stworzyć konstruktor bezargumentowy, a dodałeś już do klasy konstruktor pobierający jakieś argumenty, to konstruktor bezargumentowy będziesz musiał napisać samodzielnie.
- Jeśli tylko możesz, twórz w swoich klasach konstruktory bezargumentowe, gdyż pomogą one programistom w tworzeniu działających obiektów. Określaj domyślne wartości składowych.
- Przeciążenie konstruktorów oznacza stworzenie w danej klasie więcej niż jednego konstruktora.
- Przeciążone konstruktory muszą mieć różne listy argumentów.
- W jednej klasie nie mogą istnieć dwa konstruktory o takich samych listach argumentów. Na listę argumentów składa się ich kolejność oraz ich typy.
- Składowe zawsze uzyskują wartości domyślne, nawet jeśli nie zostaną jawnie przypisane. Domyślne wartości dla typów podstawowych to 0, 0.0 oraz false, a dla odwołań — null.

Konstruktory przeciążone

Zaostrz ołówek

Dopasuj wywołanie `new Kaczka()` z konstruktorem wywoływanym podczas tworzenia nowego obiektu. Najprostszą część zadania wykonaliśmy za Ciebie.

```
public class TestKaczki {  
  
    public static void main(String[] arg) {  
  
        int waga = 6;  
        float gestosc = 2.1F;  
        String imie = "Donald";  
        long[] piora = {1,2,3,4,5,6};  
        boolean mozeLatac = true;  
        int szybkoscLotu = 25;  
  
        Kaczka[] k = new Kaczka[7];  
        k[0] = new Kaczka();  
        k[1] = new Kaczka(gestosc, waga);  
        k[2] = new Kaczka(imie, piora);  
        k[3] = new Kaczka(mozeLatac);  
        k[4] = new Kaczka(3.3F, szybkoscLotu);  
        k[5] = new Kaczka(false);  
        k[6] = new Kaczka(szybkoscLotu, gestosc);  
    }  
}
```

```
class Kaczka {  
    int kilo = 6;  
    float lotnosc = 2.1F;  
    String imie = "Ogólna";  
    long[] piora = {1,2,3,4,5,6,7};  
    boolean mozeLatac = true;  
    int maksSzybkosc = 25;  
  
    public Kaczka() {  
        System.out.println("Kaczka typu 1");  
    }  
    public Kaczka(boolean lotna) {  
        mozeLatac = lotna;  
        System.out.println("Kaczka typu 2");  
    }  
    public Kaczka(String i, long[] p) {  
        imie = i;  
        piora = p;  
        System.out.println("Kaczka typu 3");  
    }  
    public Kaczka(int w, float l) {  
        kilo = w;  
        lotnosc = l;  
        System.out.println("Kaczka typu 4");  
    }  
    public Kaczka(float gestosc, int maks) {  
        lotnosc = gestosc;  
        maksSzybkosc = maks;  
        System.out.println("Kaczka typu 5");  
    }  
}
```

P: Wcześniej napisaliście, że warto udostępniać konstruktor bezargumentowy, aby móc określić domyślne wartości „brakujących” argumentów, jeśli ktoś taki konstruktor wywoła. Jednak czy zdarzą się takie sytuacje, w których podanie wartości domyślnych nie jest możliwe? Czy zdarzą się sytuacje, w których nie należy definiować w klasie konstruktora bezargumentowego?

O: Tak, masz rację. Są sytuacje, w których udostępnienie konstruktora bezargumentowego nie ma sensu. Możesz się o tym przekonać, przeglądając Java API — niektóre z klas nie udostępniają konstruktorów bezargumentowych. Na przykład klasa Color reprezentuje kolor. Obiekty tej klasy są używane, między innymi, do określania lub zmiany koloru czcionek albo przycisków graficznego interfejsu użytkownika. Tworząc nowy obiekt tej klasy, ma on reprezentować konkretny kolor (no wiesz: Ekstatyczny Czerwony, Śmiertelnie czekoladowobrązowy, Pożerający-wzrok-błękit-ekranu i tak dalej). Jeśli tworzysz obiekt Color, musisz w jakiś sposób określić kolor.

```
Color c = new Color(3,45,200);
```

(W tym przypadku używamy trzech liczb typu int określających wartości RGB. Sposoby wykorzystania obiektów klasy Color opiszymy w dalszej części książki, w rozdziale poświęconym bibliotece Swing). Jaki kolor uzyskałbyś, gdybyś nie podał żadnych argumentów? Twórcy Java API mogli zdecydować, że w przypadku wywołania konstruktora bezargumentowego tworzony jest piękny odcień koloru fiołkoworóżowego. Jednak dobry gust zwyciężył.

Jeśli spróbujesz utworzyć nowy obiekt klasy Color bez podawania argumentów:

```
Color c = new Color();
```

komplikator się oburzy, gdyż nie będzie w stanie znaleźć w klasie Color odpowiedniego konstruktora bezargumentowego.

```
Wiersz poleceń  
ColorTester.java:5: cannot resolve symbol  
symbol : constructor Color()  
location: class java.awt.Color  
    Color c = new Color();  
1 error
```

Nanoprzegląd. Cztery rzeczy o konstruktorach, które należy zapamiętać

- ① Konstruktor to kod wykonywany, gdy ktoś używa operatora `new` i poda po nim nazwę klasy:

```
Kaczka k = new Kaczka();
```

- ② Konstruktor musi mieć tę samą nazwę co klasa i nie może deklarować wartości wynikowej jakiegokolwiek typu:

```
public Kaczka (int wielkosc) { }
```

- ③ Jeśli nie stworzysz w klasie żadnego konstruktora, kompilator doda do niej konstruktor domyślny. Zawsze będzie to konstruktor bezargumentowy:

```
public Kaczka() { }
```

- ④ W klasie można utworzyć więcej niż jeden konstruktor, o ile każdy z nich będzie mieć inną listę argumentów. Zdefiniowanie w klasie więcej niż jednego konstruktora oznacza, że konstruktory zostały przeciążone:

```
public Kaczka() { }

public Kaczka(int wielkosc) { }

public Kaczka(String imie) { }

public Kaczka(String imie, int wielkosc) { }
```

Wykazano, że wykonywanie wszystkich zadań oznaczonych jako „Wysil szare komórki” powoduje 42 procentowy przyrost wielkości neuronów. A sam wiesz, że mówi się: „Duże neurony to...”

WYSIL SZARE KOMÓRKI

A co z klasami bazowymi?

Czy tworząc obiekt Pies, należy także wykonywać konstruktor klasy Psowate?

Jeśli klasa bazowa jest abstrakcyjna, to czy w ogóle powinna mieć konstruktor?

Tymi zagadnieniami zajmiemy się na kilku kolejnych stronach. Jak na razie przerwij lekturę na chwilę i zastanów się nad implikacjami wykorzystania konstruktorów i klas bazowych.

Nie istnieja
głupie pytania

P: Czy konstruktory muszą być publiczne (oznaczane modyfikatorem `public`)?

O: Nie. Konstruktory mogą być **publiczne**, **prywatne**, mogą także mieć dostęp **domyślny** (czyli nie trzeba ich poprzedzać żadnym modyfikatorem dostępu). Więcej informacji na temat dostępu **domyślnego** podamy w rozdziale 16. oraz w dodatku B.

P: W jaki sposób konstruktor prywatny w ogóle może być do czegoś przydatny? Przecież nikt nigdy nie może go wywołać, zatem nikt i nigdy nie będzie w stanie utworzyć nowego obiektu.

O: To nie do końca słuszne stwierdzenie. Oznaczenie czegoś modyfikatorem **private** nie oznacza wcale, że *nikt* nie może uzyskać do tego dostępu. Oznacza to, że *nikt spoza klasy* nie może uzyskać dostępu. Możemy się założyć, że myślisz, że to „paragraf 22”. Jedynie kod należący do tej samej klasy co prywatny konstruktor może stworzyć obiekt tej klasy, jednak w jaki sposób ktokolwiek może wykonać ten kod bez wcześniejszego utworzenia obiektu danej klasy? W jaki sposób, można się „dostać” do jakiegokolwiek kodu takiej klasy? *Cierpliwości*. Zajmiemy się tym w następnym rozdziale.

Chwileczkę... nigdy nie zastanawialiśmy się nad klasami bazowymi, dziedziczeniem oraz nad tym, jaki to wszystko ma związek z konstruktorami

I tu zaczyna się robić zabawnie. Czy pamiętasz tę część poprzedniego rozdziału, w której pisaliśmy o obiekcie Snowboard, który „otacza” swoją wewnętrzną część odpowiadającą klasie Object wchodzącej w skład klasy Snowboard? Najważniejsze w tym zagadnieniu było to, iż każdy obiekt zawiera nie tylko swoje *własne*, zadeklarowane składowe, lecz także *wszystkie składowe z klas bazowymi* (co w minimalnym przypadku oznacza składowe klasy Object, gdyż jest to kasa bazowa *wszystkich* innych klas w Javie).

A zatem, kiedy obiekt jest tworzony (co następuje w wyniku użycia operatora **new**; użycie tego operatora i podanie po nim nazwy klasy jest bowiem jedynym sposobem utworzenia obiektu), przydzielany jest mu obszar niezbędny do zapisania *wszystkich* składowych zadeklarowanych w klasie obiektu oraz we wszystkich klasach bazowych aż do samego wierzchołka drzewa dziedziczenia. Pomyśl o tym przez chwilę... Klasa bazowa może mieć metody ustawiające, zarządzające dostępem do składowej prywatnej. Jednak składowa ta musi gdzieś *istnieć*. To prawie tak, jak gdyby podczas tworzenia obiektu pojawiało się *wiele* nowych obiektów — ten tworzony oraz po jednym obiekcie dla każdej klasy bazowej. Jednak, z pojęciowego punktu widzenia, lepszym rozwiązaniem jest wyobrażenie sobie procesu tworzenia obiektu w sposób przedstawiony na poniższym rysunku, na którym tworzony obiekt składa się z warstw reprezentujących kolejne klasy bazowe.

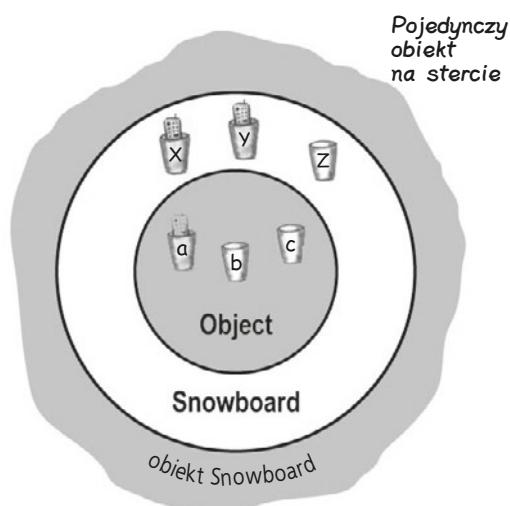
Object
Pmc a; int b; int c;
equals() getClass() hashCode() toString()

↑

Snowboard
Pmc x Pmc y int z
skrecaj() jedz() skacz() stracPanowanie()

Obiekt zawiera składowe, do których dostęp można uzyskać za pośrednictwem metod. Składowe te są tworzone podczas tworzenia obiektu klasy bazowej. (Na tym rysunku nie przedstawiono PRAWDZIWYCH składowych klasy Object, jednak nie ma to dla nas znaczenia, gdyż i tak nie mamy do nich bezpośredniego dostępu).

Także klasa Snowboard ma własne składowe, a zatem, aby utworzyć obiekt Snowboard, potrzebne będzie miejsce na składowe obu klas.



W tym przypadku na stercie tworzony jest tylko jeden obiekt, obiekt Snowboard. Jednak składa się on z dwóch części — pierwsza z nich odpowiada klasie Snowboard, a druga klasie Object. W obiekcie muszą się znaleźć wszystkie składowe z obu klas.

Znaczenie konstruktorów klas bazowej w życiu obiektu

Podczas tworzenia nowego obiektu muszą zostać wykonane konstruktory ze wszystkich klas bazowych tworzących drzewo dziedziczenia.

Dobrze to zapamiętaj.

Oznacza to, że każda klasa bazowa ma konstruktor (ponieważ każda klasa ma konstruktor) oraz że konstruktor każdej z klas należących do hierarchii dziedziczenia jest wywoływany podczas tworzenia obiektu klasy potomnej.

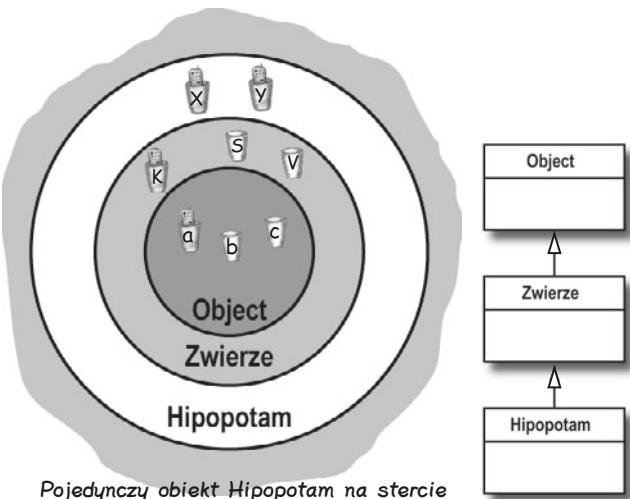
Użycie operatora **new** jest „czymś ważnym”. Operacja ta rozpoczęta reakcję łańcuchową konstruktorów. A... i jeszcze jedno... także klasy abstrakcyjne mają konstruktory. Choć nigdy nie można użyć operatora **new** do stworzenia obiektu klasy abstrakcyjnej, to jednak nawet taka klasa jest klasą bazową, zatem także jej konstruktor zostanie wywołany podczas tworzenia obiektu konkretnej klasy potomnej.

Konstruktory klas bazowych są wywoływane w celu stworzenia fragmentów obiektu odpowiadających obiektom klas bazowych. Pamiętaj, że klasa potomna może dziedziczyć metody, których działanie zależy od stanu klasy bazowej (czyli od wartości składowych klasy bazowej). Aby obiekt mógł być w pełni ukształtowany, wszystkie jego części odpowiadające obiektom klas bazowych także muszą zostać w pełni ukształtowane.

Właśnie z tego powodu konieczne jest wykonanie konstruktorów wszystkich klas bazowych. Wszystkie składowe wszystkich klas tworzących drzewo dziedziczenia muszą zostać zadeklarowane i zainicjalizowane. Nawet gdyby klasa *Zwierze* miała składowe, których klasa *Hipopotam* nie dziedziczy (na przykład składowe prywatne), to i tak klasa *Hipopotam* zależy do metod klasy *Zwierze*, które tych składowych *używają*.

Kiedy zostaje uruchomiony konstruktor, natychmiast wywołuje on konstruktor klasy bazowej, po czym zostają wywołane konstruktory kolejnych klas bazowych, aż do klasy *Object*.

Na kilku kolejnych stronach znajdziesz informacje o tym, w jaki sposób wywoływane są konstruktory klas bazowych oraz jak można wywoływać je samodzielnie. Dowiesz się także, co należy zrobić, gdy konstruktor klasy bazowej wymaga przekazania argumentów.



Nowy obiekt *Hipopotam* JEST także obiektem *Zwierze* oraz obiektem *Object*. Chcąc utworzyć obiekt *Hipopotam*, musisz także utworzyć te jego części, które odpowiadają obiektom klas *Zwierze* i *Object*.

Wszystkie te operacje tworzą proces nazywany łańcuchowym wywoływaniem konstruktorów.

Tworzenie obiektu Hipopotam oznacza także stworzenie jego fragmentów odpowiadających obiektem Zwierze oraz Object

```
public class Zwierze {
    public Zwierze() {
        System.out.println("Tworzenie obiektu Zwierze");
    }
}

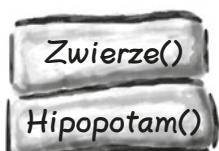
public class Hipopotam extends Zwierze {
    public Hipopotam() {
        System.out.println("Tworzenie obiektu Hipopotam");
    }
}

public class HipopotamTester {
    public static void main(String[] args) {
        System.out.println("Zaczynamy...");
        Hipopotam h = new Hipopotam();
    }
}
```

- 1 Kod jakiejś innej klasy wywołuje `new Hipopotam()`, konstruktor klasy Hipopotam jest umieszczany na ramce na szczytce stosu.



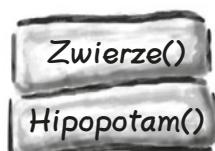
- 2 Konstruktor `Hipopotam()` wywołuje konstruktor klasy bazowej, co powoduje, że w ramce na szczytce stosu jest umieszczany konstruktor `Zwierze()`.



- 3 Konstruktor `Zwierze()` wywołuje konstruktor klasy bazowej, co sprawia, że w ramce na szczytce stosu zostaje umieszczony konstruktor `Object()`, gdyż klasa `Object` jest klasą bazową klasy `Zwierze`.



- 4 Konstruktor `Object()` zostaje wykonany, a jego ramka usunięta ze stosu. Realizacja programu wraca do konstruktora `Zwierze()` i jest kontynuowana od wiersza znajdującego się bezpośrednio poniżej wywołania konstruktora klasy bazowej.



Jakie wyniki zostaną wygenerowane? Na podstawie kodu przedstawionego z lewej strony podaj, które z poniższych wyników zostaną wygenerowane po uruchomieniu programu `HipopotamTester`. Czy będą to wyniki A czy B?

(odpowiedź jest na dole tej strony)

A

```
Wiersz polecenia
T:\>java HipopotamTester
Zaczynamy...
Tworzenie obiektu Zwierze
Tworzenie obiektu Hipopotam
```

B

```
Wiersz polecenia
T:\>java HipopotamTester
Zaczynamy...
Tworzenie obiektu Hipopotam
Tworzenie obiektu Zwierze
```

Jak można wywołać konstruktor klasy bazowej?

Zakładając, że klasa Kaczka rozszerza klasę Zwierze, mógłbyś przypuszczać, że konstruktor `Zwierze()` należy wywołać gdzieś w konstruktorze klasy Kaczka. Jednak wywoływanie konstruktora klasy bazowej działa w inny sposób:

```
public class Kaczka3 extends Zwierze {
    int wielkosc;
    public Kaczka3(int nowaWlksc) {
        Zwierze(); ← NIE TAK!
        wielkosc = nowaWlksc;
    }
}
```

Źle!

Jedynym sposobem wywołania konstruktora klasy bazowej jest użycie wywołania `super()`. Tak — `super()` wywołuje *konstruktor klasy bazowej*.

Jakie są zalety tego rozwiązania?

```
public class Kaczka extends Zwierze {
    int wielkosc;

    public Kaczka(int nowaWlksc) {
        super(); ← wystarczy napisać super()
        wielkosc = nowaWlksc;
    }
}
```

Wywołanie `super()` w konstruktorze sprawia, że na samym szczycie stosu zostanie umieszczony konstruktor klasy bazowej. A jak myślisz, co zrobi konstruktor klasy bazowej? Wywoła konstruktor swojej klasy bazowej. I tak dalej, aż na szczycie stosu znajdzie się konstruktor klasy Object. Kiedy wykonywanie konstruktora `Object()` zostanie zakończone, jego ramka jest usuwana ze stosu, a tym samym, na jego wierzchołku znajdzie się kolejny element (czyli konstruktor klasy potomnej, która wywołała konstruktor `Object()`). Następnie kończy się wykonywanie i *tego* konstruktora — proces ten jest kontynuowany aż w pewnej chwili na szczycie stosu znajdzie się początkowy konstruktor; teraz także i *jego* realizacja może się zakończyć.

A jak to się dzieje, że radziliśmy sobie bez wykonywania tych czynności?

Zapewne się domyślasz.

Nasz stary, dobry przyjaciel — kompilator — sam dodawał wywołania `super()`, jeśli myślimy o nich zapomniaeli.

A zatem, kompilator ingeruje w proces tworzenia obiektów na dwa sposoby:

- Jeśli programista *nie stworzy konstruktora sam*.

W takim przypadku kompilator doda do klasy konstruktor o następującej postaci:

```
public NazwaKlasy () {
    super();
}
```

- Jeśli programista *stworzy konstruktor, lecz nie umieści w nim wywołania super()*.

Kompilator doda wywołanie `super()` w każdym z przeciążonych konstruktorów*. Wywołanie to będzie mieć następującą postać:

```
super();
```

To wywołanie zawsze wygląda w przedstawiony sposób. Wywołanie `super()` automatycznie dodawane przez kompilator, zawsze jest wywołaniem bezargumentowym. Jeśli klasa dysponuje przeciążonymi konstruktorami, to i tak zostanie wywołany wyłącznie konstruktor bezargumentowy.

* Chyba że konstruktor wywołuje inny przeciążony konstruktor (taką sytuację możesz zobaczyć nieco dalej w tym rozdziale).

Czy dziecko może istnieć przed rodzicami?

Jeśli potraktujemy klasę bazową jako swoistego rodzica dla klas potomnych, to bez trudu będziesz mógł określić, który z obiektów musi powstać wcześniej. *Część obiektu odpowiadająca obiektemu klasy bazowej musi być w pełni ukształtowana (stworzona w całości), zanim będzie można utworzyć fragmenty obiektu odpowiadające klasie potomnej.* Pamiętaj, że działanie obiektu klasy potomnej może zależeć od elementów odziedziczonych po klasie bazowej, a wcześniejsze utworzenie tych elementów jest ważne. Na to nie można nic poradzić. Konstruktor klasy bazowej musi zostać wykonany, zanim zakończy się wykonywanie konstruktora klasy potomnej.

Spójrz jeszcze raz na rysunki przedstawiające stos zamieszczone na stronie 280. Wyraźnie widać, że choć konstruktor klasy Hipopotam jest wywoływany jako pierwszy (czyli jest pierwszym konstruktorem, którego wywołanie jest umieszczone na stosie), to jednocześnie zostaje on zakończony jako ostatni! Każdy konstruktor klasy bazowej niezwłocznie wywołuje konstruktor swojej klasy bazowej i tak się dzieje aż do momentu, gdy na szczytzie stosu znajdzie się konstruktor klasy Object. Następnie konstruktor klasy Object zostaje wykonany, a my rozpoczynamy poruszanie się ku dołowi stosu — do konstruktora klasy Zwierze. Dopiero kiedy ten konstruktor zostanie wykonany, realizacja programu wróci do konstruktora klasy Hipopotam i będzie można wykonać jego dalszą część.

Właśnie z tego powodu:

Wywołanie super() musi być pierwszą instrukcją każdego konstruktora!*



Ech... to TAKIE niesprawiedliwe. Nie ma najmniejszej możliwości, abym urodził się przed moimi rodzicami. To po prostu fatalne.

Możliwe konstruktory klasy Smok

public Smok() {
 super();
}

Te konstruktory są poprawne, gdyż programista jawnie dodat do nich wywołanie super(), umieszczając je jako pierwszą instrukcję każdego z nich.

public Smok(int i) {
 super();
 wielkosc = i;
}

public Smok() {
}

Te konstruktory są poprawne, gdyż kompilator doda do nich wywołanie super(), umieszczając je jako pierwszą instrukcję każdego z nich.

public Smok(int i) {
 wielkosc = i;
}

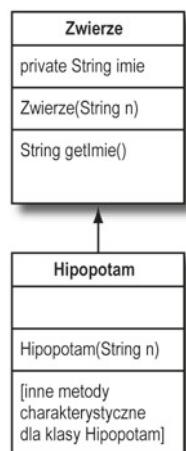
public Smok(int i) {
 wielkosc = i;
 super();
}

BŁĄD!! Tego konstruktora nie da się skompilować! Nie można jawnie umieścić wywołania super() inaczej niż jako pierwszej instrukcji w konstruktorze!

* Jest jeden wyjątek od tej reguły, o którym dowiesz się na stronie 284.

Konstruktory klas bazowych pobierające argumenty

A co, jeśli konstruktor klasy bazowej wymaga przekazania argumentów? Czy w wywołaniu `super()` można coś przekazać? Oczywiście. Gdybyś nie mógł, to nigdy nie byłbyś w stanie rozszerzyć klasy, która nie ma konstruktora bezargumentowego. Wyobraź sobie następujący scenariusz: wszystkie zwierzęta mają jakieś imiona. W klasie `Zwierze` dostępna jest metoda `getImie()` zwracająca wartość składowej `imie`. Składowa ta jest prywatna, jednak klasa potomna (w naszym przypadku będzie to klasa `Hipopotam`) dziedziczy metodę `getImie()`. I oto mamy problem — klasa `Hipopotam` dysponuje metodą `getImie()`, lecz nie ma składowej `imie`. Obiekt `Hipopotam` musi zatem polegać na swojej części odpowiadającej obiekowi `Zwierze`. Jeśli chodzi o zarządzanie imieniem — to ona jest odpowiedzialna za przechowywanie składowej i zwracanie jej wartości, kiedy ktoś wywoła metodę `getImie()` obiektu `Hipopotam`. Ale... w jaki sposób imię zostanie przekazane do tej części obiektu `Hipopotam`? Jedynym sposobem odwołania się obiektu `Hipopotam` do tej jego części, która odpowiada obiekowi `Zwierze`, jest wywołanie `super()`. A zatem to właśnie za pomocą tego wywołania obiekt `Hipopotam` może przekazać imię do tej części samego siebie, która odpowiada obiekowi `Zwierze`, i która może zapisać przekazaną wartość w składowej `imie`.



```

public abstract class Zwierze {
    private String imie;           ← Wszystkie zwierzęta
                                    (włącznie z klasami
                                    potomnymi) mają imiona.

    public String getImie() {      ← Metoda zwracająca, która
                                    odziedziczy klasa Hipopotam.
        return imie;
    }

    public Zwierze(String imieZwk) {
        imie = imieZwk;           ← Konstruktor pobierający imię
                                    zwierzęcia i zapisujący je
                                    w składowej.
    }
}

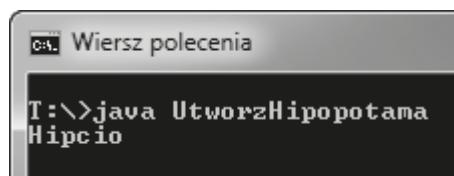
public class Hipopotam extends Zwierze {
    public Hipopotam(String imie) { ← Konstruktor klasy Hipopotam
                                    wymaga podania imienia.
        super(imie);
    }
}

public class UtworzHipopota {
    public static void main(String[] args) {
        Hipopotam h = new Hipopotam("Hipcio");
        System.out.println(h.getImie());
    }
}
  
```

Annotations:

- `Wszystkie zwierzęta (włącznie z klasami potomnymi) mają imiona.`
- `Metoda zwracająca, która odziedziczy klasa Hipopotam.`
- `Konstruktor pobierający imię zwierzęcia i zapisujący je w składowej.`
- `Konstruktor klasy Hipopotam wymaga podania imienia.`
- `i przekazuje je do konstruktora klasy Zwierze.`
- `Tworzymy obiekt Hipopotam, przekazując imię „Hipcio” w wywołaniu konstruktora. Następnie wywołujemy dziedziczoną metodę getImie().`

Moja cząstka będąca obiektem `Zwierze` musi znać moje imię, zatem pobieram je w moim własnym konstruktorze klasy `Hipopotam`, a następnie przekazuję w wywołaniu `super()`.



Wywoływanie jednego przeciążonego konstruktora z poziomu innego

A co zrobić w sytuacji, gdy dysponujemy przeciążonymi konstruktorami, które za wyjątkiem obsługi argumentów różnych typów wykonują identyczne czynności? Doskonale wiesz, że nie chcesz powielać tego samego kodu w każdym z konstruktorów (bo przysparza to problemów z utrzymaniem programu itp.), chciałbyś zatem umieścić fragment kodu używanego w konstruktorach (włącznie z wywołaniem super()) tylko w jednym z nich. Chciałbyś, aby pierwszy wywołany konstruktor, niezależnie od tego, który to będzie, wywołał „prawdziwy konstruktor” i aby to właśnie on dokonał proces tworzenia obiektu. Nie prostszego, wystarczy użyć wywołania this(). Albo this(lancuch). Albo this(27, x). Innymi słowy, wyobraź sobie, że słowo kluczowe this jest odwołaniem do **bieżącego obiektu**.

Wyrażenia this() można używać wyłącznie w konstruktorach, a co więcej, wywołanie this() musi być pierwszą instrukcją konstruktora!

Ale to powoduje pewien problem, nieprawdaż? Wcześniej napisaliśmy, że wywołanie super() także musi być pierwszą instrukcją konstruktora. No cóż, oznacza to, że musisz dokonać wyboru.

Każdy konstruktor może zawierać wywołanie this() lub super(), jednak nie może zawierać obu tych wywołań jednocześnie!

```
class Mini extends Samochod {  
  
    Color kolor;  
  
    public Mini() {  
        this(Color.red); ←  
    }  
  
    public Mini(Color k) {  
        super("Mini"); ←  
        kolor = k;  
        // dalsza inicjalizacja  
    }  
  
    public Mini(int wielkoscM) {  
        this(Color.red); ←  
        super(wielkoscM); ←  
    }  
}
```

Konstruktor bezargumentowy określa domyślny kolor i wywołuje przeciążony „prawdziwy” konstruktor (ten, który z kolei wywołuje super()).

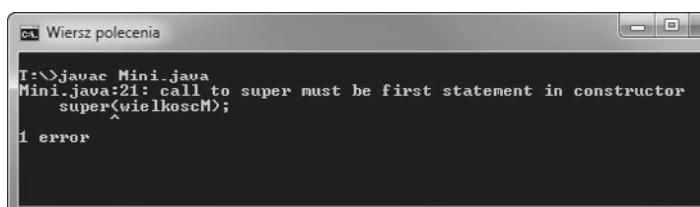
To jest „prawdziwy” konstruktor, który wykonuje „prawdziwe” zadanie inicjalizacji obiektu (włącznie z wywołaniem super()).

To nie przejdzie!!! W tym samym konstruktorze nie można umieszczać wywołań super() i this(), gdyż każde z nich musi być pierwszą instrukcją konstruktora.

Użyj wywołania this(), aby wywołać konstruktor z poziomu innego przeciążonego konstruktora należącego do tej samej klasy.

Wywołanie this() może być stosowane wyłącznie wewnątrz konstruktora i musi być jego pierwszą instrukcją.

Konstruktor może zawierać wywołanie super() LUB this(), ale nie może zawierać obu tych wywołań jednocześnie.





Zaostrz ołówek

Niektórych konstruktorów klasy SynalekBoo nie można skompilować. Sprawdź, czy będziesz w stanie określić, które z nich nie są poprawne. Dopasuj błędy zgłoszone przez kompilator z konstruktorami, które je spowodowały, w tym celu narysuj linię łączącą błędny konstruktor z odpowiednimi wynikami.

```
public class Boo {
    public Boo(int i) {}
    public Boo(String s) {}
    public Boo(String s, int i) {}
}
```

```
class SynalekBoo extends Boo {
    public SynalekBoo() {
        super("Boo");
    }

    public SynalekBoo(int i) {
        super("Fred");
    }

    public SynalekBoo(String s) {
        super(42);
    }

    public SynalekBoo(int i, String s) {
    }

    public SynalekBoo(String a, String b, String c) {
        super(a,b);
    }

    public SynalekBoo(int i, int j) {
        super("facet", j);
    }

    public SynalekBoo(int i, int x, int y) {
        super(i,"start");
    }
}
```

Zapamiętaj to dobrze

Róże są czerwone i pokłuć Cię mogą
rodzice rodzą się wcześniej, na dłucho przed Tobą.

Fragmenty obiektu odpowiadające obiektom klas bazowych muszą być w pełni utworzone zanim będzie mógł powstać obiekt klasy potomnej. Na tej samej zasadzie nie jest możliwe, abyś urodził się przed swoimi rodzicami.

Java używa przekazywania przez wartością wątki, wait(), notify()
Wykop kota

Wiersz polecenia

```
T:\>javac SynalekBoo.java
SynalekBoo.java: : cannot resolve symbol
symbol  : constructor Boo <java.lang.String,java.lang.String>
```

Wiersz polecenia

```
T:\>javac SynalekBoo.java
SynalekBoo.java: : cannot resolve symbol
symbol  : constructor Boo <int,java.lang.String>
```

Wiersz polecenia

```
T:\>javac SynalekBoo.java
SynalekBoo.java: : cannot resolve symbol
symbol  : constructor Boo <>
```

Teraz wiemy już, jak obiekt powstaje. Ale jak długo istnieje?

Istnienie obiektu zależy jedynie od istnienia odwołań do tego obiektu. Jeśli odwołanie jest uważane za „aktywne”, obiekt wciąż będzie istnieć na stercie. Jeśli odwołanie „umrze” (a już niebawem dowiesz się, co to oznacza), to „umrze” także i obiekt.

A zatem, jeśli istnienie obiektu zależy od istnienia zmiennej referencyjnej, to jak długo istnieje zmienność?

To zależy od tego, czy jest ona zmienią lokalną czy też składową obiektu. Przedstawiony poniżej przykład przedstawia „życie” zmiennej lokalnej. W tym przypadku jest to zmienność typu podstawowego, jednak czas istnienia zmiennej lokalnej jest taki sam, niezależnie od tego, czy jest to zmienność typu podstawowego czy też zmienność referencyjna.

```
public TestIstnienia1 {
    public void odczytaj() {
        int s = 42; ←
        spij(); ← Zasięg zmiennej „s” ogranicza się do metody odczytaj(),
        } → zatem nie można jej używać nigdzie poza nią.
        public void spij() { ←
            s = 7; ← BŁĄD!! Użycie zmiennej „s” w tym miejscu nie jest dozwolone!
        }
    }
}
```

Metoda spij() nie „widzi” zmiennej „s”. Zmienność ta nie znajduje się w ramach metody spij() (na stosie), zatem metoda nic o niej nie wie.

- 1 **Zmienność lokalna istnieje wyłącznie wewnętrz metody, w której została zadeklarowana.**

```
public void odczytaj() {
    int s = 42;
    // Zmienność "s" może być używana
    // wyłącznie wewnętrz tej metody.
    // Gdy metoda się zakończy,
    // zmienność całkowicie zniknie.
}
```

Zmienność „s” może być używana wyłącznie wewnętrz metody *odczytaj()*. Innymi słowy, **zasięg zmiennej ogranicza się do metody, w której została zadeklarowana**. Kod umieszczony w innych metodach tej samej klasy (lub w innych klasach) nie ma dostępu do zmiennej „s”.

- 2 **Składowa istnieje tak długo jak obiekt. Jeśli obiekt istnieje, to istnieją także jego składowe.**

```
public class Istnienie {
    int wielkosc;

    public void setWielkosc(int s) {
        wielkosc = s;
        // zmienność "s" znika wraz z
        // zakończeniem metody, jednak
        // składowa "wielkosc" może być
        // używana w dowolnym miejscu
        // klasy
    }
}
```

Zmienność „s” (tym razem jest to parametr metody) istnieje wyłącznie w czasie wykonywania metody *setWielkosc()*, jednak składowa *wielkosc* będzie istnieć tak długo, jak długo istnieje *obiekt*, a nie jedynie w czasie wykonywania *metody*.

Różnica pomiędzy czasem **istnienia** oraz **zasięgiem** zmiennych lokalnych:

Czas istnienia

Zmienna lokalna *istnieje* dopóty, dopóki na stosie znajduje się ramka jej metody. Innymi słowy, zmienna lokalna istnieje *do momentu zakończenia metody*.

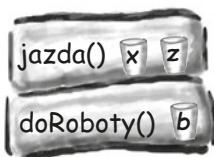
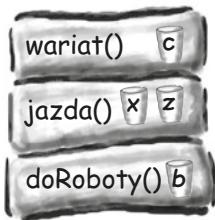
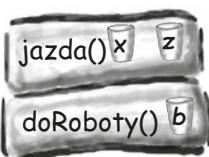
Zasięg

Zmienna lokalna istnieje wyłącznie w *zasięgu* metody, wewnętrzko której została zadeklarowana. Kiedy metoda ta wywoła inną metodę, to zmienna wciąż będzie istnieć, jednak nie będzie dostępna aż do wznowienia wykonywania „jej” metody. *Zmiennej można używać wyłącznie w zasięgu, w którym jest dostępna.*

```
public void doRoboty() {
    boolean b = true;
    jazda();
}

public void jazda(int x) {
    int z = x + 24;
    wariat();
    // tu jest więcej kodu
}

public void wariat() {
    char c = 'a';
}
```



- 1 Ramka metody `doRoboty()` jest umieszczana na stosie. Zmienna „`b`” istnieje i jest dostępna w bieżącym zasięgu.

- 2 Na wierzchołku stosu jest umieszczana ramka metody `jazda()`. Zmienne „`x`” oraz „`z`” istnieją i są dostępne w bieżącym zasięgu; z kolei zmienna „`b`” istnieje, lecz nie jest dostępna.

- 3 Na wierzchołku stosu jest umieszczana ramka metody `wariat()` i aktualnie zmienna „`c`” istnieje i jest dostępna w bieżącym zasięgu. Pozostałe trzy zmienne istnieją, lecz nie są dostępne.

- 4 Metoda `wariat()` zostaje zakończona, a jej ramka usunięta ze stosu, dlatego też zmienna „`c`” przestaje istnieć i nie jest dostępna w bieżącym zasięgu. Kiedy realizacja metody `jazda()` zostanie wznowiona w miejscu, w którym wcześniej ją wstrzymano, zmienne „`x`” oraz „`y`” nie tylko będą istnieć, lecz także staną się dostępne w bieżącym zasięgu. Zmienna „`b`” wciąż będzie istnieć, lecz nie będzie dostępna (aż do momentu zakończenia metody `jazda()`).

W czasie gdy zmienna istnieje, jej stan jest zachowywany. Na przykład, tak długo jak ramka metody `doRoboty()` znajduje się na stosie, zmienna „`b`” zachowuje swoją wartość. Jednak zmiennej tej można używać wyłącznie wtedy, gdy ramka metody `doRoboty()` znajduje się na wierzchołku stosu. Innymi słowy, zmiennych lokalnych można używać *jedynie* wtedy, gdy metoda, w której zostały one zadeklarowane, jest aktualnie wykonywana (a nie, gdy czekamy na zakończenie innych metod, których ramki znajdują się wyżej na stosie).

A co ze zmiennymi referencyjnymi?

Zmienne referencyjne obowiązują te same zasady co zmienne typów podstawowych. Zmienna referencyjna może być używana wyłącznie w zasięgu, w jakim jest dostępna. A to oznacza, że nie można używać pilota do obiektu, jeśli nie mamy dostępu do zmiennej referencyjnej. Jednak prawdziwe pytanie brzmi:

„W jaki sposób istnienie zmiennej wpływa na istnienie obiektu?”

Obiekt istnieje, dopóki istnieją aktywne odwołania do niego. Jeśli zmienna referencyjna przestanie być dostępna w bieżącym zasięgu, lecz wciąż istnieje, to obiekt, do którego zmienna ta się odwołuje, wciąż będzie istnieć na stercie. Jednak musisz się o to zapytać... „Co się dzieje, gdy ramka metody przechowująca odwołanie zostanie usunięta ze stosu po zakończeniu metody?”

Jeśli to było *jedyne* aktywne odwołanie do obiektu, to obiekt pozostaje porzucony na stercie. Zmienna referencyjna została zniszczona wraz z ramką, a zatem porzucony obiekt *oficjalnie* stał się... grzanką. Cała sztuka polega na tym, aby wiedzieć, w którym momencie obiekt zostaje *uznany za nadający się do odśmiecenia*.

Kiedy obiekt zostanie uznany za nadający się do odśmiecenia, nie będziesz już musiał przejmować się odzyskiwaniem zajętej przez niego pamięci. Kiedy programowi zacznie brakować pamięci, odśmiecacz usunie część lub wszystkie obiekty przeznaczone do odśmiecenia i w ten sposób zapobiegnie problemom. Oczywiście może się zdarzyć, że programowi zabraknie pamięci, jednak *nie* nastąpi to wcześniej niż po „wyrzuceniu do śmieci” wszystkich obiektów, które zostały uznane za nadające się do odśmiecenia. Twoim zadaniem jest upewnienie się, że obiekty zostaną porzucone (co sprawi, że będzie można je uznać za nadające się do usunięcia), kiedy nie będziesz już ich potrzebować, przez co odśmiecacz będzie mógł coś z nimi zrobić. Jeśli jednak będziesz się „kurczowo trzymać” swoich obiektów, to odśmiecacz nie będzie w stanie Ci pomóc, a Ty staniesz w obliczu ryzyka, że Twój program zginie z powodu braku pamięci.

Istnienie obiektu nie ma żadnej wartości, żadnego znaczenia ani celu, chyba że ktoś będzie mieć odwołanie do tego obiektu.

Jeśli obiekt jest nieosiągalny, to nie możesz go poprosić o wykonanie jakiejkolwiek czynności; w takiej sytuacji obiekt jest po prostu jedną wielką stratą bitów.

Jeśli obiekt nie jest osiągalny, to odśmiecacz dowie się o tym. Wcześniej czy później obiekt przestanie istnieć.



Obiekt zostaje uznany za nadający się do odśmiecenia, kiedy przestanie istnieć ostatnie aktywne odwołanie wskazujące na niego.

1 Trzy sposoby pozbycia się odwołania do obiektu:

Zasięg, w jakim dostępne jest odwołanie, zakończy się.

```
void jazda() {  
    Istnienie i = new Istnienie();  
}
```

Odwołanie „i” przestaje istnieć wraz z końcem metody.

2 W odwołaniu zostanie zapisany inny obiekt.

```
Istnienie i = new Istnienie();  
i = new Istnienie();
```

Pierwszy obiekt jest porzucony, gdy odwołanie „i” zostaje „przeprogramowane” na nowy obiekt.

3 Odwołaniu zostanie jawnie przypisana wartość null.

```
Istnienie i = new Istnienie();  
i = null;
```

Pierwszy obiekt jest porzucony, gdy „z pilota zostanie usunięte oprogramowanie”.

Zabójca obiektów nr 1

Zasięg, w jakim dostępne jest odwołanie, zakończy się.



```
public class StosRef {
    public void bor() {
        wor();
    }

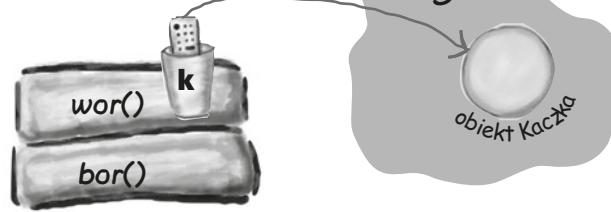
    public void wor() {
        Kaczka k = new Kaczka();
    }
}
```



- 1 Na stosie jest umieszczana ramka metody `bor()`, nie są deklarowane żadne zmienne.



- 2 Na stosie zapisywana jest ramka metody `wor()`. Metoda deklaruje jedną zmienną referencyjną i tworzy jeden obiekt, który jest w tej zmiennej zapisywany. Obiekt jest tworzony na stercie, a odwołanie jest aktywne i dostępne w bieżącym zakresie.



Nowy obiekt klasy `Kaczka` zostaje umieszczony na stercie, a odwołanie „`k`” będzie aktywne i dostępne w bieżącym zasięgu do momentu zakończenia realizacji metody `wor()`. Obiekt `Kaczka` jest uznawany za aktywny.

- 3 Metoda `wor()` zostaje zakończona, a jej ramka jest usuwana ze stosu. Sprawia to, że zmienna „`k`” staje się niedostępna. Realizacja programu wraca do metody `bor()`, jednak metoda ta nie może korzystać ze zmiennej „`k`”.



Ojejku! Ponieważ zmienna „`k`” „odeszła”, a ramka metody `wor()` została usunięta ze stosu, obiekt `Kaczka` zostaje porzucony. Odśmiecacz wkracza do akcji.

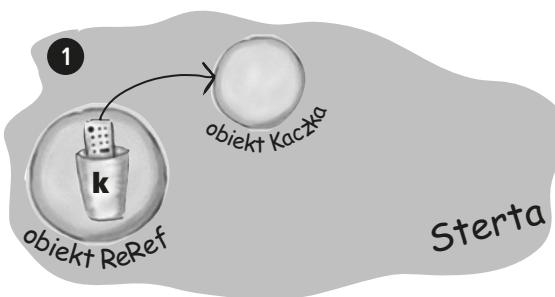
Zabójca obiektów nr 2

Zapisanie w odwołaniu innego obiektu

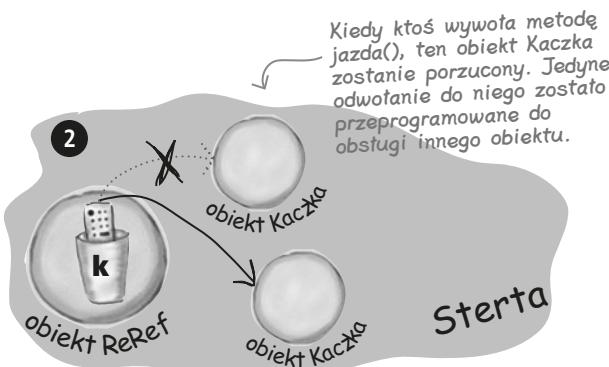
```
public class ReRef {  
  
    Kaczka k = new Kaczka();  
  
    public void jazda() {  
        k = new Kaczka();  
    }  
}
```



Rany! A wszystko co trzeba było zrobić, to usunąć odwołanie. Chyba jeszcze wtedy nie znali zarządzania pamięcią.



Na stercie tworzony jest nowy obiekt Kaczka, a odwołanie do niego jest zapamiętywane w składowej „k”. Ponieważ „k” jest składową, zatem obiekt Kaczka będzie istnieć tak długo jak obiekt ReRef, który go stworzył. Chyba że...



W składowej „k” jest zapisywany nowy obiekt Kaczka, przez co oryginalny (pierwszy) obiekt zostaje porzucony. Teraz ten pierwszy obiekt do niczego się nam już nie przyda.



Zabójca obiektów nr 3

**Jawne przypisanie
odwołaniu
wartości null**



```
public class ReRef {
    Kaczka k = new Kaczka();

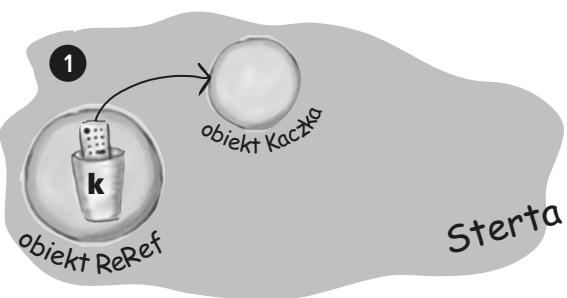
    public void jazda() {
        k = null;
    }
}
```

Znaczenie null

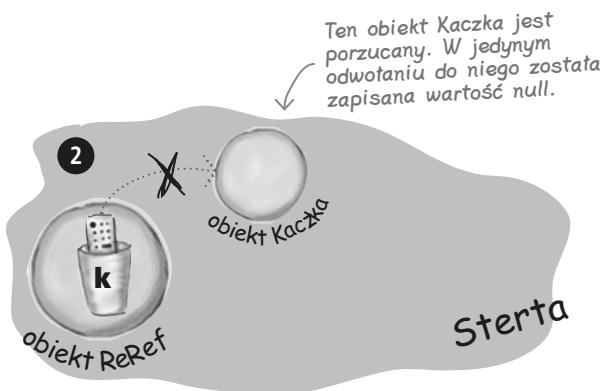
Przypisanie odwołaniu wartości **null** można porównać z usunięciem oprogramowania z pilota. Innymi słowy, mamy pilot, lecz nie mamy telewizora, którym mógłby on sterować. Takie „puste” odwołanie zawiera bity reprezentujące wartość **null** (*nie wiemy, co to są za bity, ani nas to nie obchodzi, o ile tylko JVM wie*).

W naszym realnym świecie, kiedy naciskamy przyciski na niezaprogramowanym pilocie, po prostu nic się nie dzieje. Jednak w Javie nie można „naciskać przycisków” (czyli używać operatora kropki) „puстego” odwołania, gdyż wirtualna maszyna Java wie (i jest to zagadnienie związane z realizacją programu, a nie komplikacją kodu), że oczekujesz szczekania, choć nie ma żadnego psa, który mógłby wykonywać Twoje polecenia!

Jeśli spróbujesz wykonać jakąś operację przy wykorzystaniu „puстego” odwołania, podczas działania programu zostanie zgłoszony wyjątek `NullPointerException`. W rozdziale 11. dowiesz się wszystkiego na temat wyjątków.



Na stercie jest tworzony nowy obiekt Kaczka, a odwołanie do niego jest zapisywane w składowej „k”. Ponieważ „k” jest składową, zatem obiekt Kaczka będzie istnieć tak długo, jak obiekt ReRef, który go stworzyt. Chyba że...



W składowej „k” jest zapisywana wartość **null**, co można porównać z pilotem, który nie jest zaprogramowany do obsługi żadnego urządzenia. Dopóki składowa ta nie zostanie ponownie zaprogramowana (czyli do momentu, gdy zapiszemy w niej jakiś obiekt), nie możemy jej nawet używać wraz z operatorem kropki.

Pogawędkи при kominku



Składowa

Chciałabym zacząć, ponieważ zazwyczaj jestem ważniejsza dla programu niż zmienna lokalna. Istnieje, aby wspierać obiekt i to zazwyczaj, przez całe jego życie. Czymże w końcu jest obiekt bez *stanu*? A czymże jest stan? **To wartości przechowywane w składowych.**

Ależ nie zrozum mnie źle, doskonale rozumiem, jaką rolę odgrywasz w metodach, chodzi tylko o to, że istniejesz tak krótko. Jesteś taka „chwilowa”. To pewnie dlatego nazywają was „zmiennymi chwilowymi”.

Och, przepraszam. Doskonale to rozumiem.

Nigdy o tym nie myślałam w taki sposób. Co robicie, kiedy te inne metody są wykonywane, a wy czekacie aż wasza ramka ponownie znajdzie się na wierzchołku stosu?

Temat dzisiejszej pogawędki: Składowa oraz zmienna lokalna rozmawiają o życiu i śmierci (w sposób nad wyraz uprzejmy).

Zmienna lokalna

Doceniam twój punkt widzenia, jak również doceniam znaczenie stanu obiektu, jednak nie chciałabym, aby czytelnicy byli wprowadzani w błąd. Zmienne lokalne są *naprawdę* ważne. Parafrując twoją wypowiedź: „Czymże jest obiekt bez *działania*?” A czym jest działanie? Algorytmami w metodach. I możesz się założyć o swoje bity, że jakieś *zmienne lokalne* będą konieczne do działania tych algorytmów.

W społeczności zmiennych lokalnych określenie „zmienna chwilowa” jest uważane za uwłaczające. Wolimy takie określenia jak „lokalna”, „stosowa”, „automatyczna” lub „ograniczona zasięgiem”.

Jednak to prawda, że nie istniejemy zbyt długo, co gorsze, nasze życie nie jest także szczególnie *dobre*. Po pierwsze jesteśmy upychane w jednej ramce na stosie ze wszystkimi innymi zmiennymi lokalnymi. Potem, jeśli metoda, w której istniejemy, wywoła inną metodę, to nad nami wstawiana jest kolejna ramka. A jeśli ta *inna* metoda wywoła jeszcze *następną* metodę... i tak dalej. Czasami musimy czekać całe wieki, żeby te wszystkie metody nad nami się zakończyły i aby nasza metoda mogła być dalej wykonywana.

Nic. Absolutnie nic. To tak jakby się było w hibernacji — tym czymś, co robią ludziom w filmach fantastycznonaukowych, gdy trzeba odbywać bardzo długie podróże. Taka zatrzymana animacja, poważnie. Po prostu tam siedzimy i czekamy. Dopóki nasza ramka istnieje, jesteśmy bezpieczne, my i nasze wartości. Jednak moment, gdy nasza metoda ponownie zacznie być wykonywana jest zarazem dobry jak i zły. Z jednej strony ponownie stajemy się aktywne, natomiast z drugiej strony zegar odmierzający nasze krótkie życia znowu zaczyna tykać. Im więcej razy działanie naszej metody jest przerywane, tym bardziej zbliżamy się do jej zakończenia. A wszyscy wiemy, co wtedy następuje.

Składowa

Tak, kiedyś widzieliśmy edukacyjny film wideo na ten temat. Koniec faktycznie wygląda dosyć brutalnie. Mam na myśli to, że gdy realizacja metody dotrze do jej zamykającego nawiąsu klamrowego, ramka jest dosłownie *zrzucana ze stosu*!

To musi naprawdę boleć.

Ja żyję na stercie, razem z obiekta mi. A w zasadzie to nie z obiekta mi, a w obiekcie. W obiekcie, którego stan przechowuję. Muszę zaznaczyć, że życie na stercie może być całkiem wygodne. Wiele z nas ma poczucie winy, zwłaszcza w okresie wakacji.

W porządku, hipotetycznie. Gdybym była składową obiektu Obroza i cały ten obiekt zostałby odśmiecony, to faktycznie wszystkie jego składowe zostałyby potraktowane jak wszystkie niepotrzebne pudełka po pizzy. Jednak słyszałam, że prawie nigdy tak się nie dzieje.

Ale czy pozwolą nam się *napić*?

Zmienna lokalna

Jasne, jeszcze mi o tym przypominaj. W informatyce używamy zwrotu *zdejmowana*, na przykład: „ramka została zdjęta ze stosu”. To może brzmieć zabawnie albo jak gdyby to był jakiś sport ekstremalny. Ale cóż, sama widziałaś, jaka jest prawda. Może zatem porozmawiamy o tobie? Wiem jak wygląda moje życie na stosie, ale gdzie ty żyjesz?

Jednak nie *zawsze* życie tak długo jak obiekt, w którym zostałyście zadeklarowane, nieprawdaż? Założmy, że jest obiekt Pies zawierający składową Obroza. Wyobraź sobie, że to ty jesteś składową obiektu Obroza, na przykład odwołaniem do obiektu Spraczka lub jakiegoś innego, i żyjesz sobie szczęśliwie w obiekcie Obroza, który żyje sobie szczęśliwie w obiekcie Pies. Co się jednak stanie, jeśli pies będzie chciał mieć nową obrozę lub zapisze wartość null w składowej przechowującej odwołanie do obiektu Obroza? W takim przypadku obiekt Obroza zostanie uznany za nadający się do odśmieczenia. Zatem... jeśli będziesz składową istniejącą wewnątrz obiektu Obroza i cały ten obiekt zostanie porzucony, co się w takim przypadku z tobą stanie?

I ty w to wierzysz? Tak mówią, żebyśmy były zmotywowane i zachowały wysoką produktywność. Ale czy nie zapominasz o czymś innym? Co by się stało, gdybyś była składową obiektu, do którego odwołanie jest przechowywane *wyłącznie* w zmiennej *lokalnej*? Jeśli to ja będę jedynym odwołaniem do obiektu, w którym istniejesz, to kiedy ja zginę, zginiesz razem ze mną. Niezależnie od tego, czy ci się to podoba czy nie, nasze losy mogą się splatać. Zatem zapomnijmy o tym wszystkim i chodźmy lepiej się upić... póki jeszcze możemy. Jak to się mówi: „Carpe RAM”.



BĄDŹ odśmiecaczem



Który z wierszy kodu podanych z prawej strony po dodaniu do kodu przedstawionego z lewej strony w punkcie oznaczonym literą „A” spowoduje, że dokładnie jeden dodatkowy obiekt zostanie uznany za nadający się do odśmieczenia? (Założ, że kod w miejscu oznaczonym A, i poniższe wywołania kolejnych metod, będzie realizowany dłużej, przez co odśmiecacz będzie mógł wykonać swoje zadanie).

```
public class Odsmiecacz {  
    public static Odsmiecacz doRoboty() {  
        Odsmiecacz nw0dsm = new Odsmiecacz();  
        doRoboty2(nw0dsm);  
        return nw0dsm;  
    }  
  
    public static void main(String[] args) {  
        Odsmiecacz odsm1;  
        Odsmiecacz odsm2 = new Odsmiecacz();  
        Odsmiecacz odsm3 = new Odsmiecacz();  
        Odsmiecacz odsm4 = odsm3;  
        odsm1 = doRoboty();  
  
        // wywołania kolejnych metod  
    }  
  
    public static void doRoboty2(Odsmiecacz kopia0dsm) {  
        Odsmiecacz lokalny0dsm;  
    }  
}
```

A

- 1 kopia0dsm = null;
- 2 odsm2 = null;
- 3 nw0dsm = odsm3;
- 4 odsm1 = null;
- 5 nw0dsm = null;
- 6 odsm4 = null;
- 7 odsm3 = odsm2;
- 8 odsm1 = odsm4;
- 9 odsm3 = null;



Ćwiczenie

Popularne obiekty

```

class Pszczoly {
    Miodek[] miodki;
}

class Szop {
    Kubelek k;
    Miodek md;
}

class Kubelek {
    Miodek md;
}

class Mis {
    Miodek miodzio;
}

public class Miodek {
    public static void main(String[] args) {
        Miodek spodeczek = new Miodek();
        Miodek[] miód = {spodeczek, spodeczek, spodeczek, spodeczek};
        Pszczoly p1 = new Pszczoly();
        p1.miodki = miód;
        Mis[] ms = new Mis[5];
        for (int x=0; x < 5; x++) {
            ms[x] = new Mis();
            ms[x].miodzio = spodeczek;
        }
        Kubelek k = new Kubelek();
        k.md = spodeczek;
        Szop s = new Szop(); ← Oto nowy obiekt Szop!
        ↑
        s.md = spodeczek;
        s.k = k;
        k = null;
    } // koniec main
}

```

Oto zmienna referencyjna „s”.

W przedstawionym przykładowym programie tworzonych jest kilka nowych obiektów. Twoim zadaniem jest odnalezienie „najbardziej popularnego” obiektu, czyli obiektu, na który wskazuje najwięcej odwołań. Następnie podaj, ile jest odwołań do tego obiektu i gdzie się one znajdują! Sami zacznijmy od wskazania jednego nowego obiektu i skojarzonej z nim zmiennej referencyjnej.

Powodzenia!

Ćwiczenie: Zagadka na pięć minut



Zagadka na pięć minut



„Robiliśmy symulację już cztery razy i zawsze temperatura modułu głównego spadała poniżej poziomu nominalnego”, powiedziała zdesperowana Sara. „W zeszłym tygodniu zainstalowaliśmy nowe czujniki temperatury. Odczyty z czujników klimatyzatorów, zaprojektowanych do chłodzenia pomieszczeń mieszkalnych, wydają się mieścić w granicach określonych w specyfikacji, dlatego nasze analizy koncentrują się na grzejnikach, które mają ogrzewać pomieszczenia.” Tomek kiwnął głową. Początkowo wydawało się, że nanotechnologia pozwoli im zakończyć pracę na długo przed terminem. Teraz, kiedy do końca terminu zostało tylko pięć tygodni, niektóre z podstawowych funkcji podtrzymywania życia stacji orbitalnej wciąż nie przechodziły testów symulacyjnych.

— Jakie współczynniki symulujesz? — zapytał Tomek.

— Cóż, o ile dobrze rozumiem, do czego zmierzasz, to już o tym pomyśleliśmy — odpowiedziała Sara. — Kontrola misji nie zaakceptuje systemu o tak kluczowym znaczeniu, jeśli nie zmieścimy się w granicach określonych w specyfikacji. Mamy uruchamiać jednostki symulacyjne klimatyzatorów v3 i v2 w stosunku 2:1 — ciągnęła Sara. — Ogólny współczynnik grzejników do klimatyzatorów ma wynosić 4:3.

— A co ze zużyciem energii Saro? — spytał Tomek. Sara zawałała się przez moment. — Cóż, to kolejny problem. Zużycie energii jest wyższe niż przewidywano. Zespół już sprawdza, co się dzieje. Jednak ponieważ elementy nanowymiarowe są bezprzewodowe, trudno jest określić, ile energii zużywają grzejniki, a ile klimatyzatory.

— Ogólne współczynniki zużycia energii — kontynuowała Sara — wynoszą 3:2, przy czym to klimatyzatory mają pobierać więcej energii z sieci bezprzewodowej.

— Dobrze, Saro — powiedział Tomek — rzućmy okiem na kod inicjalizujący symulację. Musimy znaleźć ten problem i to szybko!

```
import java.util.*;
class KlimatyzatorV2 {
    KlimatyzatorV2(ArrayList lista) {
        for(int x=0; x<5; x++) {
            lista.add(new JednSym("KlimatyzatorV2"));
        }
    }
}
class KlimatyzatorV3 extends KlimatyzatorV2 {
    KlimatyzatorV3(ArrayList lglista) {
        super(lglista);
        for (int g=0; g<10; g++) {
            lglista.add(new JednSym("KlimatyzatorV3"));
        }
    }
}
class Grzejnik {
    Grzejnik(ArrayList glista) {
        glista.add(new JednSym("Grzejnik"));
    }
}
```

Zagadki na pięć minut ciąg dalszy...

```

public class SPZTester {
    public static void main(String[] args) {
        ArrayList lista = new ArrayList();
        KlimatyzatorV2 v2 = new KlimatyzatorV2(lista);
        KlimatyzatorV3 v3 = new KlimatyzatorV3(lista);
        for(int z=0; z<20; z++) {
            Grzejnik grz = new Grzejnik(lista);
        }
    }
}

class JedenSym {
    String typUrz;

    JedenSym(String typ) {
        typUrz = typ;
    }

    int zuzycieEnergi() {
        if ("Grzejnik".equals(typUrz)) {
            return 2;
        } else {
            return 4;
        }
    }
}

```

Tomek rzucił okiem na kod i na jego ustach pojawił się niewielki uśmiechek. — Chyba znalazłem błąd, Saro, i założę się, że wiem również, o ile procent twoje odczyty zużycia energii różnią się od zaplanowanych!

Co Tomek podejrzewał? W jaki sposób mógł odgadnąć błędy w odczytach zużycia energii i jakich kilka wierszy kodu można by dodać, aby ułatwić testowanie tego programu?



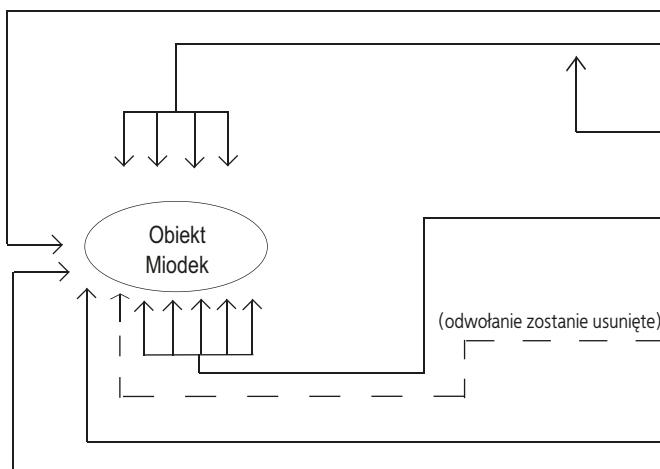
Ćwiczenie rozwiązywanie

Odśmiecacz

1. kopiaOdsm = null; Nie — ten wiersz kodu próbuje użyć zmiennej, która nie jest dostępna w bieżącym zakresie.
2. odsm2 = null; Tak — odsm2 to jedyna zmienna referencyjna odwołująca się do tego obiektu.
3. nwOdsm = odsm3; Nie — kolejna zmienna, która nie jest dostępna w bieżącym zakresie.
4. odsm1 = null; Tak — odsm1 zawiera jedyne odwołanie, gdyż nwOdsm jest poza zakresem.
5. nwOdsm = null; Nie — nwOdsm jest poza zakresem.
6. odsm4 = null; Nie — odsm3 wciąż odwołuje się do tego samego obiektu.
7. odsm3 = odsm2; Nie — odsm4 wciąż odwołuje się do tego samego obiektu.
8. odsm1 = odsm4; Tak — instrukcja usuwa jedyne odwołanie do obiektu.
9. odsm3 = null; Nie — odsm4 wciąż odwołuje się do tego samego obiektu.

Popularne obiekty

Prawdopodobnie nietrudno było ustalić, że bezwzględnie „najpopularniejszym” obiektem w przedstawionej klasie był obiekt Miodek, do którego początkowo odwoływała się zmienna spodeczek. Zapewne nieco trudniejsze było zauważenie, że wszystkie odwołania klasy Miodek wskazują **ten sam obiekt!** Bezpośrednio przed zakończeniem metody main() istnieje aż 12 aktywnych odwołań do tego obiektu. Przez pewien czas aktywne jest także odwołanie k.md, jednak później składowej k zostaje przypisana wartość null. Ponieważ składowa s.k wciąż odwołuje się do obiektu Kubelek, zatem składowa s.k.md także odwołuje się do tego samego obiektu Miodek (choć nigdzie nie została jawnie zadeklarowana).



```

public class Miodek {
    public static void main(String[] args) {
        Miodek spodeczek = new Miodek();
        Miodek[] miod = { spodeczek, spodeczek,
                          spodeczek, spodeczek};
        Pszczoly p1 = new Pszczoly();
        p1.miodki = miod;
        Mis[] ms = new Mis[5];
        for (int x=0; x < 5; x++) {
            ms[x] = new Mis();
            ms[x].miodzio = spodeczek;
        }
        Kubelek k = new Kubelek();
        k.md = spodeczek;
        Szop s = new Szop();
        s.md = spodeczek;
        s.k = k;
        k = null;
    } // koniec main
}

```



Zagadka na pięć minut. Rozwiążanie

Tomek zauważył, że konstruktor klasy KlimatyzatorV2 pobiera obiekt ArrayList, a to oznacza, że za każdym razem, gdy był wywoływany konstruktor klasy KlimatyzatorV3, przekazywał on obiekt ArrayList do konstruktora klasy KlimatyzatorV2 za pośrednictwem wywołania super(). A to oznacza z kolei, że tworzonych było pięć dodatkowych obiektów JednSym KlimatyzatorV2. Gdyby się okazało, że Tomek ma rację, to całkowite zużycie energii powinno wynosić 120, a nie 100, jak przewidywały współczynniki Sary.

Ponieważ klasy wszystkich urządzeń tworzyły obiekty JednSym, problem można by szybko znaleźć, tworząc w tej klasie konstruktor wyświetlający komunikat.

10. Liczby oraz metody i składowe statyczne

Liczby mają znaczenie



Wykonuj obliczenia matematyczne. Jednak operacje na liczbach to nie tylko podstawowe działania arytmetyczne. Na przykład mógłbyś chcieć wyznaczyć wartość bezwzględną liczby, zaokrąglić ją albo znaleźć większą z dwóch wartości. Mógłbyś chcieć wyświetlać liczby dokładnie z dwiema cyframi po przecinku dziesiętnym, albo oddzielać grupy cyfr w dużych liczbach aby łatwiej je było odczytać. A co z operacjami na datach? Mógłbyś chcieć wyświetlać daty na różne sposoby, a nawet wykonywać na nich *operacje*, takie jak „dodaj do dzisiejszej daty trzy tygodnie”. A co z zamienianiem łańcuchów znaków na liczby? Oraz z zamienianiem liczb na łańcuchy znaków? Masz szczęście. W Java API jest wiele łatwych i gotowych do użycia metod służących do operowania na liczbach. Jednak większość z nich to metody **statyczne**, dlatego też zaczniemy od przedstawienia, co oznacza, że metoda lub składowa jest statyczna, w tym rozważymy także zagadnienie tworzenia stałych — czyli statycznych zmiennych *finalnych*.

Metody klasy **Math** — najlepsze z możliwych odpowiedników metod globalnych

W Javie w ogóle *nie ma niczego* globalnego, ale wyobraź sobie metodę, której działanie nie zależy od wartości jakichkolwiek składowych. Na przykład metodę `round()` klasy `Math`. Metoda ta za każdym razem wykonuje tę samą operację — zaokrąglą liczbę zmiennoprzecinkową (przekazaną jako argument) do najbliższej liczby całkowitej. Za każdym razem. Gdybyś miał 10 tysięcy obiektów klasy `Math` i dla każdego z nich wywołał metodę `round(42.2)`, to uzyskałbyś wartość 42. Zawsze. Innymi słowy, metoda operuje na argumencie, a jej działanie nie zależy do stanu składowych obiektu. Jedyną wartością, jaka zmienia sposób działania metody `round()`, jest przekazany do niej argument!

Czy tworzenie obiektu klasy `Math` tylko po to, aby wykonać metodę `round()`, nie wydaje się niepotrzebnym marnowaniem przestrzeni na stercie? Poza tym, co z innymi metodami klasy `Math`, takim jak `min()`, która pobiera dwie liczby jednego z typów podstawowych i zwraca mniejszą z nich? Albo `max()`. Albo `abs()` — metoda, która zwraca wartość bezwzględną liczby.

Żadna z tych metod nigdy nie wykorzystuje wartości składowych.

Właściwie klasa `Math` *nie ma żadnych składowych*. A zatem stworzenie obiektu tej klasy nic by nam nie dało. I wiesz co? Nie musisz tworzyć takiego obiektu. Właściwie to nawet nie możesz.

Jeśli spróbujesz stworzyć obiekt klasy `Math`:

```
Math obiektMath = new Math();
```

Kompilator zgłosi błąd:

```
Wiersz polecenia
T:>javac TestKlasyMath.java
TestKlasyMath.java:16: Math() has private access in java.lang.Math
    Math obiektMath = new Math();
                           ^
1 error
T:>
```

Metody klasy `Math` nie wykorzystują wartości składowych. A ponieważ są to metody „statyczne”, zatem aby z nich korzystać, nie jest nam potrzebny obiekt `Math`. W zupełności wystarczy nam klasa `Math`.

```
int x = Math.round(42.2);
int y = Math.min(56,12);
int z = Math.abs(-423);
```



Te metody nigdy nie wykorzystują wartości składowych, dlatego też nie muszą nic wiedzieć o żadnym konkretnym obiekcie.

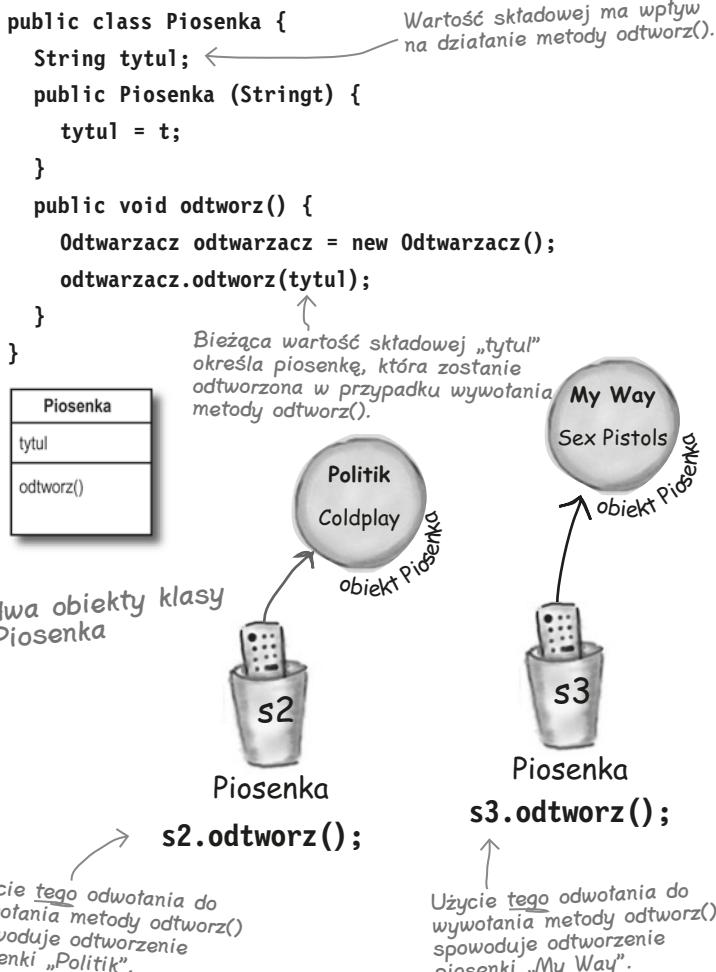


Ten błąd informuje, że konstruktor klasy `Math` jest prywatny! Oznacza to, że nigdy nie będziesz mógł użyć operatora `new` do stworzenia nowego obiektu klasy `Math`.

Różnice pomiędzy metodami zwyczajnymi a statycznymi

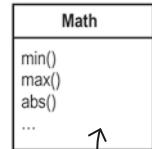
Java jest językiem zorientowanym obiektowo, jednak od czasu do czasu pojawia się szczególny przypadek (na przykład metody klasy Math), w którym posługiwanie się obiektem tej klasy nie jest konieczne. Słowo kluczowe **static** pozwala wywoływać metodę *bez konieczności posługiwania się obiektem tej klasy*. Metoda statyczna oznacza „działanie, które nie zależy od składowych, a zatem nie wymaga użycia obiektu. Wystarczy sama klasa”.

Zwyczajne („niestatyczne”) metody



Metody statyczne

```
public static int min(int a, int b)
{
    // zwraca mniejszą z dwóch liczb
}
```



Żadnych składowych. Działanie metod nie zmienia się w zależności od stanu składowych.

Math.min(42,36);

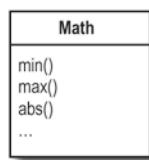
Użyj nazwy klasy zamiast zmiennej referencyjnej.



ŻADNYCH OBIEKTÓW!!!

ŻADNYCH OBIEKTÓW na tym rysunku!
Ani jednego!!!

Metody statyczne wywołuje się, podając nazwę klasy



```
Math.min(88,86);
```

Normalne — czyli „niestatyczne” — metody wywołuje się, podając nazwę zmiennej referencyjnej



```
Piosenka t2 = new Piosenka();  
t2.odtworz();
```

Co oznacza, że klasa ma statyczne metody

Często (choć nie zawsze) fakt, że klasa ma metody statyczne, oznacza, iż nie należy tworzyć obiektów tej klasy. W rozdziale 8. przedstawiliśmy klasy abstrakcyjne oraz opisaliśmy, w jaki sposób oznaczenie klasy modyfikatorem **abstract** uniemożliwia tworzenie obiektów tej klasy przy użyciu operatora new. Innymi słowy, **tworzenie obiektów klas abstrakcyjnych jest niemożliwe**.

Niemniej jednak istnieje także inny sposób na to, aby uniemożliwić tworzenie obiektów normalnych — czyli „nieabstrakcyjnych” — klas. Polega on na oznaczeniu konstruktorów klasy jako prywatne (czyli poprzedzenie ich modyfikatorem **private**). Pamiętasz, że stworzenie prywatnej *metody* oznacza, że jedynie kod należący do tej samej klasy będzie mógł z tej metody korzystać. Stworzenie prywatnego *konstruktora* oznacza w zasadzie to samo — jedynie kod należący do tej samej klasy będzie mógł wywołać konstruktor. Nikt spoza klasy nie będzie mógł użyć operatora new, aby utworzyć jej obiekt. Właśnie w ten sposób działa klasa Math. Jej konstruktor jest prywatny i nie możemy tworzyć nowych obiektów tej klasy. Kompilator wie, że Twój kod nie ma dostępu do prywatnego konstruktora.

Nie oznacza to wcale, że nigdy nie należy tworzyć obiektów klasy, która ma jedną lub kilka metod statycznych.

W rzeczywistości każda klasa, w której zostanie umieszczona metoda main(), jest klasą z metodą statyczną!

Zazwyczaj metoda main() tworzona jest po to, aby można było uruchomić i przetestować inną klasę, i prawie zawsze odbywa się to poprzez utworzenie nowego obiektu tej klasy oraz wywoływanie jego metod.

Zatem nic nie stoi na przeszkodzie, aby w jednej klasie umieszczać metody statyczne i normalne, choć stworzenie choćby jednej normalnej metody oznacza, że będzie musiał istnieć jakiś sposób stworzenia obiektu danej klasy. Jedynym sposobem utworzenia nowego obiektu jest użycie operatora new lub mechanizmu deserializacji (lub wykorzystania specjalnego mechanizmu „refleksji” — Java Reflection API — którym nie będziemy zajmować się w tej książce). Nie istnieją żadne inne metody. Jednak ciekawe może być zagadnienie, *kto* używa operatora new; zajmiemy się nim w dalszej części rozdziału.

Metody statyczne nie mogą używać niestatycznych składowych!

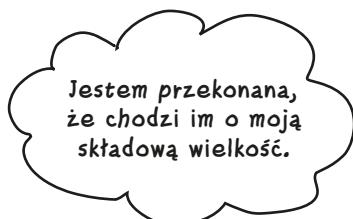
Metody statyczne działają, nie wiedząc nic o jakimkolwiek konkretnym obiekcie klasy, do której należą. A, jak się przekonałeś na poprzednich stronach, może się zdarzyć, że obiekt takiej klasy w ogóle nie będzie istnieć. Ponieważ metoda statyczna jest wywoływana przy użyciu nazwy klasy (`Math.random()`), a nie odwołania do obiektu (`t2.odtworz()`), zatem metoda taka nie może odwoływać się do składowych klasy. Metoda taka nie wie bowiem, z którego obiektu należy odczytać wartości składowych.

Jeśli spróbujesz skompilować następujący fragment kodu:

```
public class Kaczka {
    private int wielkosc;
    public static void main(String[] args) {
        System.out.println("Ta kaczka ma rozmiar: " + wielkosc);
    }
    public void setWielkosc(int a) {
        wielkosc = a;
    }
    public int getWielkosc() {
        return wielkosc;
    }
}
```

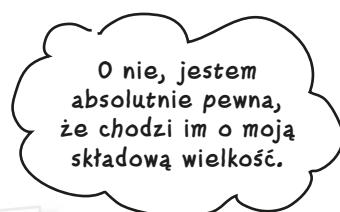
Która Kaczka?
Czyja wielkość?

Jeśli gdzieś na stercie
jest obiekt Kaczka,
to my nic o nim nie
wiemy.



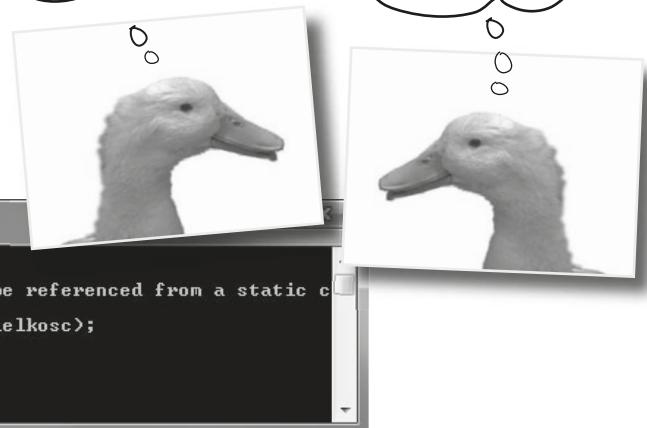
Jeśli spróbujesz użyć składowej wewnętrz metody statycznej, kompilator pomyśli sobie: „Nie wiem, o składowe którego obiektu Ci chodzi!”. Nawet, jeśli na stercie znajdzie się dziesięć obiektów Kaczka,

to metoda statyczna nie będzie wiedzieć o żadnym z nich.



Zostanie wyświetlony poniższy błąd:

```
Wiersz polecenia
T:>javac Kaczka.java
Kaczka.java:6: non-static variable wielkosc cannot be referenced from a static context
    System.out.println("Ta kaczka ma rozmiar: " + wielkosc);
               ^
1 error
T:>
```



Metody statyczne nie mogą także wywoływać metod niestatycznych!

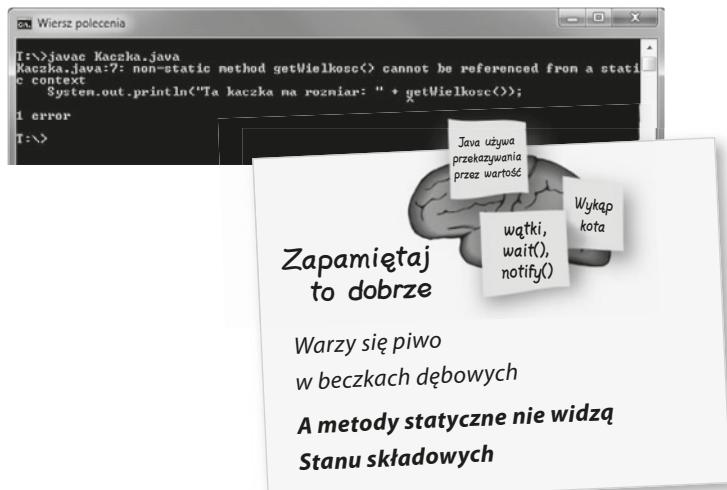
Co robią metody „normalne”, czyli niestatyczne? *Zazwyczaj wykorzystują stan składowych i na jego podstawie modyfikują swoje działanie.* Metoda `getImie()` zwraca wartość składowej `imie`. Ale czyjej składowej? Oczywiście obiektu użytego do wywołania tej metody.

Tego nie uda się skompilować:

```
public class Kaczka {  
    private int wielkosc;  
  
    public static void main(String[] args) {  
        System.out.println("Ta kaczka ma rozmiar: " +  
            getWielkosc());  
    }  
  
    public void setWielkosc(int a) {  
        wielkosc = a;  
    }  
  
    public int getWielkosc() {  
        return wielkosc; ←  
    }  
}
```

Wywołanie metody `getWielkosc()` jedynie na chwilę odsuwa nieuniknione — metoda ta wykorzystuje składową `wielkosc`.

I wracamy do tego samego problemu... czyja składową odczytujemy?



Nie istnieją głupie pytania

P: Co się stanie, jeśli z poziomu metody statycznej spróbujesz wywołać metodę niestatyczną, lecz nie będzie się w niej używać żadnych składowych? Czy kompilator na to pozwoli?

O: Nie. Kompilator wie, że niezależnie od tego, czy to zrobisz czy nie, to jednak możesz te składowe wykorzystać. I pomyśl o implikacjach... jeśli byłbyś w stanie skompilować kod napisany w taki sposób, to co by się stało, gdybyś pewnego dnia zapragnął zmienić implementację tak, aby jednak korzystała ze składowych? Lub co gorsze, co by się stało gdyby klasa potomna *przesłoniła* tę niestatyczną metodę i wewnętrznie niej wykorzystała składowe?

P: Móglbym przysiąc, że widziałem kod, który wywołuje metodę statyczną, używając do tego celu nie nazwy klasy, lecz odwołania.

O: To można zrobić, jednak to tak, jak mówiła Ci mama: „To, że coś jest legalne, wcale nie oznacza, że to jest dobre”. Choć można wywołać metodę statyczną, używając w tym celu dowolnego obiektu klasy, to jednak takie rozwiązanie jest mylące (i przyczynia się do powstawania mniej czytelnego kodu). Możesz napisać:

```
Kaczka k = new Kaczka();  
String[] s = {};  
k.main(s);
```

Powyższy fragment kodu jest poprawny, lecz kompilator i tak przetworzy to odwołanie na prawdziwą nazwę klasy („W porządku, `k` jest typu `Kaczka`, a metoda `main()` jest statyczna, zatem wywołam statyczną metodę `main()` dostępną w klasie `Kaczka`”). Innymi słowy, użycie zmiennej `k` nie oznacza wcale, że metoda `main()` uzyska jakąkolwiek szczególną wiedzę na temat obiektu, do którego zmienna ta się odwołuje. Jest to jedynie alternatywny sposób wywołania metody statycznej, jednak nie zmienia to faktu, że metoda jest statyczna!

Składowa statyczna — ta sama wartość we wszystkich obiektach danej klasy

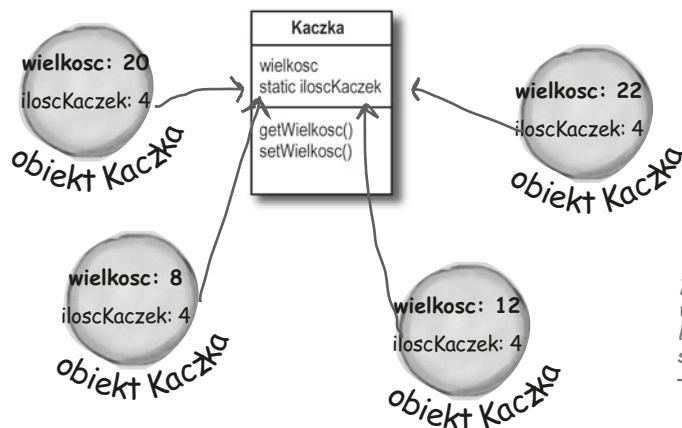
Wyobraź sobie, że chciałbyś policzyć, ile obiektów Kaczka zostało stworzonych podczas działania programu. W jaki sposób można by to zrobić? Może wykorzystać składową inkrementowaną w konstruktorze klasy?

```
class Kaczka {
    int liczbaKaczek = 0;
    public Kaczka() {
        liczbaKaczek++;
    }
}
```

Ta instrukcja przypisze składowej liczbaKaczek wartość 1 za każdym razem, gdy zostanie utworzony nowy obiekt Kaczka.

Zauważ, że to rozwiązanie nie będzie działać, gdyż liczbaKaczek jest składową i dla każdego obiektu Kaczka przyjmuje początkową wartość 0. Mogłybyś spróbować wywoływać metodę jakieśiej innej klasy, ale to już jest dziwne i nieeleganckie rozwiązanie. Potrzebujesz klasy, która ma tylko jedną kopię składowej — klasy, której wszystkie obiekty będą wspólnie używały tej jednej kopii składowej.

Właśnie taką możliwość zapewniają składowe statyczne — zawierają wartość, która jest wspólnie wykorzystywana przez wszystkie obiekty danej klasy. Innymi słowy — jedna składowa dla całej klasy, a nie po jednej dla wszystkich obiektów.



Statyczna składowa ilosckaczek jest inicjalizowana tylko raz, podczas wczytywania klasy, a nie za każdym razem, gdy tworzone jest nowy obiekt.

```
public class Kaczka {
    private int wielkosc;
    private static int ilosckaczek = 0;

    public Kaczka() {
        ilosckaczek++;
    }

    public void setWielkosc(int a) {
        wielkosc = a;
    }

    public int getWielkosc() {
        return wielkosc;
    }
}
```

Teraz, ponieważ składowa ilosckaczek jest statyczna, jej wartość będzie inkrementowana za każdym razem, gdy zostanie wykonany konstruktor klasy Kaczka, jednocześnie utworzenie nowego obiektu nie będzie powodować zapisania w tej składowej wartości 0.

Obiekt Kaczka nie przechowuje swojej własnej kopii składowej ilosckaczek.

Ponieważ składowa ilosckaczek jest statyczna, zatem wszystkie obiekty Kaczka wspólnie używają tylko jednej kopii tej składowej. Taką składową statyczną możesz sobie wyobrazić jako zmienną, która istnieje nie w obiekcie, lecz w KLASIE.

Każdy obiekt Kaczka dysponuje własną składową wielkosc, lecz istnieje tylko jedna kopią składowej ilosckaczek — kopia dostępna w klasie.

Składowe statyczne



Składowe statyczne
są współdzielone.

Wszystkie obiekty tej samej klasy
wspólnie używają jednej kopii każdej
ze składowych statycznych.

Składowe — po jednej w każdym obiekcie.

Składowe statyczne — jedna dla całej klasy.



WYŁĘŻ UMYSŁ

We wcześniejszej części rozdziału dowiedziałeś się, że stworzenie prywatnego konstruktora oznacza, że kod zewnętrzny względem klasy nie będzie mógł tworzyć jej obiektów. Innymi słowy, obiekty klasy posiadającej prywatny konstruktor można tworzyć wyłącznie w kodzie należącym do tej klasy. (W tym przypadku pojawia się problem podobny do: „co było pierwsze — jajko czy kura?”).

A co zrobić gdybyś chciał napisać klasę pozwalającą na utworzenie tylko JEDNEGO obiektu i gdybyś chciał, aby każdy, kto zgłosi chęć użycia obiektu tej klasy, mógł posługiwać się tym samym obiektem?

Inicjalizacja składowych statycznych

Składowe statyczne są inicjalizowane w momencie wczytywania klasy. Klasa jest wczytywana, ponieważ wirtualna maszyna Javy decyduje, że nadszedł czas, aby to zrobić. Zazwyczaj klasa jest wczytywana, ponieważ ktoś spróbował utworzyć nowy obiekt tej klasy. Jako programista także i Ty masz możliwość nakazania wirtualnej maszynie Javy, aby wczytała klasę, jednak prawdopodobnie nie będziesz musiał korzystać z niej zbyt często. Niemal we wszystkich przypadkach najlepszym rozwiązańiem będzie pozostawienie decyzji, kiedy wczytać klasę, w gestii wirtualnej maszyny Javy.

Inicjalizacja statyczna gwarantuje że:

Składowe statyczne klasy są inicjowane, zanim będzie można utworzyć jakikolwiek *obiekt* tej klasy.

Składowe statyczne klasy są inicjowane, zanim zostanie wywołania jakikolwiek *statyczna metoda* tej klasy.

```
class Gracz {
    static int iloscGraczy = 0;

    private String imie;
    public Gracz(String n) {
        imie = n;
        iloscGraczy++;
    }
}

public class GraczTester {
    public static void main(String[] args) {
        System.out.println(Gracz.iloscGraczy);
        Gracz pierwszy = new Gracz("Tiger Woods");
        System.out.println(Gracz.iloscGraczy);
    }
}
```

Składowa `iloscGraczy` jest inicjalizowana w momencie wczytywania klasy. Składową tę jawnie inicjujemy wartością 0, choć nie jest to konieczne, gdyż 0 jest domyślną wartością dla składowych całkowitych.

Domyślne wartości zadeklarowanych, lecz niezainicjalowanych składowych statycznych oraz dla zwyczajnych składowych są takie same:

Dla składowych całkowitych typów podstawowych (`long`, `short`, `int` itd.) jest to wartość 0.

Dla składowych zmiennoprzecinkowych typów podstawowych (`float` oraz `double`) jest to wartość 0.0.

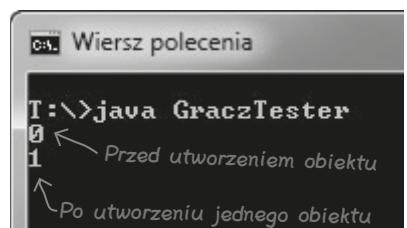
Dla składowych logicznych jest to `false`.

Dla odwołań jest to `null`.

Do składowej statycznej można się odwoływać tak samo jak do metody statycznej — poprzedzając jej nazwę nazwą klasy.

Składowe statyczne są inicjalizowane w momencie wczytywania klasy. Jeśli nie zainicjujesz składowej statycznej w sposób jawnym (poprzez przypisanie jej wartość w momencie deklarowania) zostanie jej przypisana wartość domyślna. W takim przypadku składowe całkowite przyjmują wartość 0, co oznacza, że nie musielibyśmy jawnie używać instrukcji `iloscGraczy = 0`. W przypadku, gdy zadeklarujemy składową statyczną, lecz nie określmy jej wartości, zostanie jej przypisana wartość domyślna odpowiedniego typu — dzieje się to na takiej samej zasadzie jak określanie wartości zadeklarowanych składowych.

Wszystkie składowe statyczne klasy są inicjalizowane, zanim będzie można utworzyć jakikolwiek obiekt tej klasy.



Rolę stałych pełnią statyczne zmienne finalne

Poprzedzenie zmiennej modyfikatorem **final** oznacza, że po jej zainicjowaniu już nigdy nie będzie można zmienić jej wartości. Innymi słowy, wartość statycznej zmiennej finalnej będzie pozostała taka sama tak długo, jak długo klasa będzie wczytana. Odszukaj w Java API deklarację zmiennej Math.PI, a okaże się, że wygląda ona następująco:

```
public static final double PI = 3.141592653589793;
```

Zmienna jest oznaczona modyfikatorem **public**, aby dowolny kod mógł uzyskać do niej dostęp.

Zmienna jest także oznaczona modyfikatorem **static**, aby korzystanie z niej nie wymagało tworzenia obiektu klasy Math (czego, jak zapewne pamiętasz, nie jesteśmy w stanie zrobić).

Zmienna jest oznaczona modyfikatorem **final**, ponieważ wartość PI nie zmienia się (przynajmniej z punktu widzenia Javy).

Java nie udostępnia żadnego innego sposobu oznaczania i wyróżniania „stałych”, istnieje jednak konwencja, która ułatwia określenie, czy dana zmienna jest „stałą” czy też nie.

Nazwy zmiennych, których wartości nigdy nie ulegają zmianie, są zapisywane wyłącznie wielkimi literami!

Inicjalizator statyczny to blok kodu, który jest wykonywany w momencie wczytywania klasy, zanim jakikolwiek inny kod będzie mógł z niej skorzystać.

Z tego powodu doskonale nadaje się on do inicjalizacji statycznych zmiennych finalnych.

```
class Test {  
    final static int x;  
    static {  
        x = 42;  
    }  
}
```

Aby określić wartość statycznej zmiennej finalnej

1 Podczas jej deklarowania:

```
public class Test {  
    public static final int TEST_X = 25;  
}
```

LUB Zwróć uwagę na konwencję nazewnictwa – statyczne zmienne finalne pełnią funkcję stałych, zatem ich nazwy należy zapisywać wielkimi literami, a poszczególne słowa oddzielać znakami podkreślenia.

2 W inicjalizatorze statycznym:

```
public class Bar {  
    public static final double BAR_MUZA;  
  
    static {  
        BAR_MUZA = (double) Math.random();  
    }  
}
```

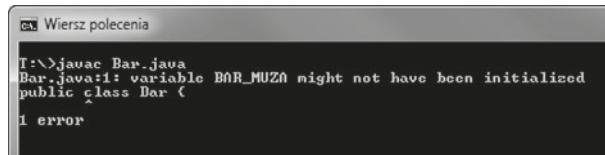
Ten kod jest wykonywany bezpośrednio po wczytaniu klasy, a przed wywołaniem jakichkolwiek metod statycznych oraz zanim będzie można użyć jakichkolwiek składowych statycznych.

Jeśli wartość zmiennej finalnej nie zostanie określona w jednym z dwóch podanych wcześniej miejsc, na przykład:

```
public class Bar {  
    public static final double BAR_MUZA;  
}
```

brak inicjalizacji!

Kompilator to wykryje i zgłosi błąd:



Nie tylko zmienne statyczne mogą być finalne

Słowa kluczowego **final** można także używać do modyfikowania właściwości zmiennych niestatycznych, zmiennych lokalnych, a nawet parametrów metod. W każdym z tych przypadków użycie słowa kluczowego **final** oznacza to samo — że wartość nie może ulec zmianie. Jednak można go także użyć, aby uniemożliwić przesłanianie metod lub tworzenie klas potomnych.

Niestatyczne zmienne finalne

```
class Blef {
    final int wielkosc = 3;   ← Teraz już nie możesz zmienić wielkości.
    final int numer;

    Blef() {
        numer = 42;   ← Teraz już nie możesz zmienić numeru.
    }

    void zrob(final int x) {
        // nie można zmieniać wartości x
    }

    void zrobJeszcze() {
        final int z = 7;
        // nie można zmieniać wartości z
    }
}
```

Metoda finalna

```
class Blef {
    final void numerKlasyczny() {
        // ważne rzeczy, których
        // nie można przesłaniać
    }
}
```

Klasa finalna

```
final class MojaDoskonałaKlasa {
    // nie można jej rozszerzać
}
```

Zmienna **finalna** oznacza,
że jej wartości nie można zmieniać.

Metoda **finalna** oznacza,
że nie można jej przesłaniać.

Klasa **finalna** oznacza,
że nie można jej rozszerzać
(czyli tworzyć klas potomnych).



Nie istniejąca
glupie pytania

P: Metoda statyczna nie może uzyskać dostępu do składowych niestatycznych, ale czy niestatyczna metoda może mieć dostęp do składowych statycznych?

O: Oczywiście. Metoda niestatyczna zawsze może wywołać metodę statyczną klasy lub skorzystać ze składowej statycznej.

P: Dlaczego mógłbym chcieć tworzyć klasy finalne? Czy to nie jest sprzeczne z podstawowym celem programowania zorientowanego obiektywem?

O: I tak i nie. Jedną z podstawowych przyczyn, dla jakich tworzy się klasy finalne, jest ich zabezpieczenie. Na przykład nie można tworzyć klas potomnych klasy `String`. Wyobraź sobie, jaki chaos zapanowałby, gdyby ktoś rozszerzył klasę `String` i, wykorzystując polimorfizm, przekazywał swoje własne obiekty zamiast oczekiwanych obiektów `String`. Jeśli musisz mieć pewność co do działania metod w jakiejś klasie, to utwórz klasę finalną.

P: Czy oznaczanie metod jako finalnych w przypadku, gdy już cała klasa została oznaczona jako finalna, nie jest nadmiarowe?

O: Jeśli klasa została oznaczona jako finalna, to oznaczanie w ten sam sposób metod, jest niepotrzebne. Zastanów się — jeśli klasa jest finalna, to nigdy nie będzie możliwa zdefiniowanie jej klas potomnych, a zatem nigdy nie będzie możliwa przełożyć żadnej z jej metod.

Jednak z drugiej strony, jeśli chcesz pozwolić innym programistom na rozszerzanie swojej klasy i chcesz dać im możliwość przesyłania jej niektórych, lecz nie wszystkich, metod, to nie oznaczaj całej klasy jako finalnej — zamiast tego oznacz jako finalne tylko jej wybrane metody. Oznaczenie metody kwalifikatorem `final` determinuje, że nie będzie jej możliwa przełożenie w klasie potomnej.



CELNE SPOSTRZEŻENIA

- **Metodę statyczną** wywołuje się przy wykorzystaniu nazwy klasy, a nie nazwy odwołania do obiektu `Math.random()` w odróżnieniu od mojaTmp.jazda().
- Metodę statyczną można wywoływać nawet w przypadku, gdy na stercie nie ma żadnych obiektów danej klasy.
- Metody statyczne doskonale nadają się na metody pomocnicze, których działanie nie zależy (i nigdy nie będzie zależeć) od stanu jakichkolwiek składowych obiektu.
- Metoda statyczna nie jest skojarzona z żadnym konkretnym obiektem (jest natomiast skojarzona z klasą) i dlatego nie może korzystać z żadnych składowych tej klasy. Nie wiedziałaby bowiem, której składowej ma użyć.
- Metoda statyczna nie może wywoływać metod niestatycznych, gdyż zazwyczaj ich działanie zależy od stanu składowych.
- Jeśli dysponujesz klasą, która zawiera wyłącznie metody statyczne, i nie chcesz pozwolić na tworzenie obiektów tej klasy, to oznacz jej konstruktor jako prywatny.
- **Składowa statyczna** to składowa, która jest wspólnie wykorzystywana przez wszystkie obiekty danej klasy. Istnieje tylko jedna kopia składowej statycznej i jest ona przechowywana w klasie, z kolei kopia składowej istnieje w każdym obiekcie danej klasy.
- Metody statyczne mogą korzystać ze składowych statycznych.
- Aby stworzyć stałą w Javie, należy oznaczyć zmienną jako statyczną i finalną.
- Statycznej zmiennej finalnej należy przypisać wartość bądź to w miejscu jej deklaracji, bądź też w statycznym inicjalizatorze:

```
static {  
    KOD_PSA = 420;  
}
```
- Stosowana konwencja nazewnicza nakazuje, aby nazwy stałych (statycznych zmiennych finalnych) były zapisywane wielkimi literami.
- Po przypisaniu wartości zmiennej finalnej nie można już jej zmienić.
- Wartość **składowej** finalnej należy określić bądź to w miejscu jej deklaracji, bądź też w konstruktorze.
- Metod finalnych nie można przesłaniać.
- Klasy finalnej nie można rozszerzać (nie można tworzyć jej klas potomnych).



Zaostrz ołówek



CO JEST POPRAWNE?

Pamiętając o wszystkich podanych informacjach dotyczących słów kluczowych final i static, określ, które z poniższych przykładów można skompilować.

1 public class Test1 {
 static int x;

```
        public void doRoboty() {  
            System.out.println(x);  
        }  
    }
```

2 public class Test2 {
 int x;

```
    public static void doRoboty() {  
        System.out.println(x);  
    }  
}
```

3 public class Test3 {
 final int x;

```
    public void doRoboty() {  
        System.out.println(x);  
    }  
}
```

4 public class Test4 {
 static final int x = 12;

 public void doRoboty() {
 System.out.println(x);
 }
}

5 public class Test5 {
 static final int x = 12;

 public void doRoboty(final int x) {
 System.out.println(x);
 }
}

6 public class Test6 {
 int x = 12;

 public static void doRoboty(final int x) {
 System.out.println(x);
 }
}

Metody klasy Math

Teraz, kiedy już wiemy, jak działają metody statyczne, przyjrzyjmy się dokładniej wybranym metodom klasy Math. Nie opisywaliśmy wszystkich metod dostępnych w tej klasie, a jedynie kilka wybranych. Szczegółowe informacje na temat wszystkich metod tej klasy, w tym także `sqrt()`, `tan()`, `ceil()`, `floor()` czy też `asin()`, można znaleźć w dokumentacji Java API.

Math.random()

Zwraca liczbę typu double z zakresu od 0.0 do 1.0 (jednak sama liczba 1.0 nie należy do zbioru możliwych wartości wynikowych).

```
double r1 = Math.random();
int r2 = (int) (Math.random() * 5);
```

Math.abs()

Zwraca liczbę typu double, która stanowi wartość bezwzględną przekazanego argumentu. Jest to metoda przeciążona, zatem jeśli w jej wywołaniu zostanie przekazana liczba typu int, metoda zwróci liczbę tego samego typu. W przypadku przekazania liczby typu double, zostanie zwrocona liczba typu double.

```
int x = Math.abs(-240); // zwróci 240
double d = Math.abs(240.45); // zwróci 240.45
```

Math.round()

Zwraca liczbę typu int lub long (zależnie od tego, czy przekazany argument jest typu float czy double) zaokrągloną do najbliższej liczby całkowitej.

```
int x = Math.round(-24.8f); // zwraca -25
int y = Math.round(24.45f); // zwraca 24
```

Pamiętaj, że literaty zmiennoprzecinkowe domyślnie są typu double, chyba że na końcu zostanie zapisana litera „f”.

Math.min()

Zwraca wartość stanowiącą minimum dwóch przekazanych argumentów. Metoda jest przeciążona i można do niej przekazywać wartości typów int, long, float oraz double.

```
int x = Math.min(24,240); // zwraca 24
double y = Math.min(90876.5, 90876.49); // zwraca 90876.49
```

Math.max()

Zwraca wartość stanowiącą maksimum dwóch przekazanych argumentów. Metoda jest przeciążona i można do niej przekazywać wartości typów int, long, float oraz double.

```
int x = Math.max(24,240); // zwraca 240
double y = Math.max(90876.5, 90876.49); // zwraca 90876.5
```

Reprezentowanie wartości typów podstawowych w formie obiektów

Czasami będziesz chciał potraktować wartość typu podstawowego jako obiekt. Na przykład przed pojawiением się wersji 5.0 Javy w języku tym nie mogłeś w bezpośredni sposób umieścić wartości typu podstawowego w kolekcji takiej jak ArrayList lub HashMap.

```
int x = 32;
ArrayList tb1 = new ArrayList();
tb1.add(x);
```

To nie zadziata, chyba że używasz Javy w wersji 5.0 lub nowszej!! W klasie ArrayList nie ma metody add(), która umożliwiałaby przekazanie liczby typu int! (Dostępna jest jedynie metoda add() pobierająca odwołania do obiektów, a nie wartości typów podstawowych).

Istnieje klasa odpowiadająca każdemu z typów podstawowych. Co więcej, wszystkie te klasy należą do pakietu `java.lang`, a zatem nigdy nie trzeba ich importować. Można je bardzo łatwo rozpoznać, gdyż ich nazwy odpowiadają nazwom typów podstawowych, przy czym zawsze zaczynają się one od wielkiej litery, aby zachować zgodność z konwencjami tworzenia nazw klas.

Och tak, z powodów, których do końca nie zna absolutnie nikt na całym świecie, projektanci API zdecydowali się stworzyć klasy, których nazwy nie będą idealnie odpowiadać nazwom *reprezentowanych* przez nie typów podstawowych. Zaraz się przekonasz, co to oznacza:

Boolean

Character

Byte

Short

Integer

Long

Float

Double

Uważaj! Te nazwy nie odpowiadają dokładnie nazwom typów podstawowych. Są to pełne stowa.

Tworzenie obiektu reprezentującego wartość

```
int i = 288;
Integer iRep = new Integer(i);
```

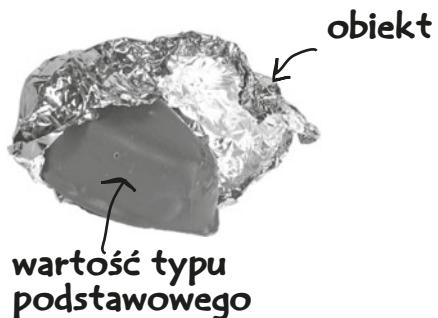
Przekaż wartość typu podstawowego w wywołaniu konstruktora klasy. O właściwie tak.

W taki sposób działają wszystkie klasy reprezentujące typy podstawowe. Klasa Boolean udostępnia metodę `booleanValue()`, klasa Character metodę `charValue()` i tak dalej.

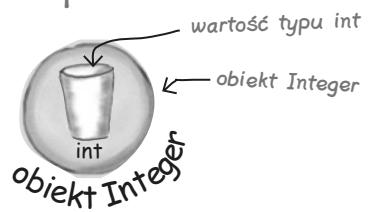
Notatka: Rysunek u góry strony przedstawia kawałek czekolady zawinięty — „opakowany” — w folię. Rozumiesz? Opakowanie. Niektórym osobom rysunek ten kojarzy się z pieczonymi ziemniakami, ale to też jest dobre porównanie.

Uzyskiwanie wartości na podstawie obiektu

```
int wtp = iRep.intValue();
```



Kiedy musisz potraktować wartość typu podstawowego jako obiekt, „opakuj” ją. Jeśli używasz Javy w wersji wcześniejszej niż 5.0, to właśnie w taki sposób będziesz musiał postępować, chcąc zapisać wartość typu podstawowego w kolekcji takiej jak ArrayList lub HashMap.





To głupie... Twierdzisz, że nie mogę stworzyć tablicy ArrayList operującej na liczbach typu int? Każdą z tych małych, głupich liczb muszę opakować w jakiś obiekt Integer? A później wypakować, kiedy będę chciał pobrać wartość z tablicy ArrayList? To przecież czysta strata czasu i możliwość popełnienia całej masy błędów...

Przed wprowadzeniem Javy 5.0 to TY musiałeś wykonywać tę pracę

Ona ma rację. We wszystkich wersjach Javy przed pojawieniem się wersji 5.0 typy podstawowe były typami podstawowymi, a odwołania do obiektów były odwołaniami do obiektów. I NIGDY nie traktowano ich zamiennie. „Opakowywanie” i „wypakowywanie” wartości typów podstawowych zawsze zależało wyłącznie od Ciebie — programisty. Nie istniał żaden sposób pozwalający na przekazanie wartości typu podstawowego w miejscu, gdzie było oczekiwane odwołanie do obiektu, podobnie jak nie można było zapisać w zmiennej typu podstawowego wartości wynikowej metody zwracającej odwołanie; i to nawet jeśli zwracane odwołanie wskazywało na obiekt Integer, a zmienna była typu int. Po prostu nie było żadnej relacji pomiędzy typami Integer i int; jedynym, co łączyło te dwa typy, był fakt, że klasa Integer posiada składową typu int (służącą do przechowywania wartości typu podstawowego). A zatem wszystkie niezbędne operacje musiałeś wykonać sam.

Tablica ArrayList zawierająca wartości typu podstawowego int

Wersja bez automatycznego konwertowania (czyli przed wprowadzeniem Javy 5.0)

```
public void listaStarymSposobem() {  
    ArrayList tablicaLiczb = new ArrayList();  
  
    tablicaLiczb.add(new Integer(3));  
  
    Integer jeden = (Integer) tablicaLiczb.get(0);  
  
    int intJeden = jeden.intValue();  
}
```

Tworzymy tablice, obiekt ArrayList. (Pamiętaj, że we wcześniejszych wersjach języka, przed wprowadzeniem Javy 5.0, można było tworzyć obiekty ArrayList bez określania TYPU ich zawartości. Dlatego też wszystkie obiekty ArrayList były tablicami obiektów typu Object.

W tablicy nie można umieścić wartości 3 typu int, dlatego trzeba zapisać ją w obiekcie Integer.

Obiekt pobierany z tablicy jest typu Object, jednak można go rzutować do typu Integer.

W końcu możesz odczytać wartość typu int z obiektu Integer.

Automatyczne konwertowanie: zacieranie granicy pomiędzy typami podstawowymi i obiektami

Mechanizm automatycznej konwersji (ang. *autoboxing*) wprowadzony w Javie 5.0 przekształca wartości typów podstawowych na odpowiednie obiekty, a co więcej — robi to *automatycznie*!

Zobaczmy, co się stanie, jeśli zechcemy wykorzystać tablicę `ArrayList` do przechowywania wartości typu `int`.

Tablica `ArrayList` zawierająca wartości typu podstawowego `int`

Wersja z automatycznym konwertowaniem (dostępna w wersji 5.0 lub wyższej języka Java)

```
public void liczbyPoNowemu() {  
    ArrayList<Integer> tablicaLiczb = new ArrayList<Integer>;  
  
    tablicaLiczb.add(3);      Po prostu dodaj wartość!  
  
    int liczba = tablicaLiczb.get(0);  
}  
Kompilator automatycznie skonwertuje obiekt typu Integer, tak byśmy mogli przypisać odczytywaną wartość bezpośrednio do zmiennej typu int, bez konieczności wywoływania metody intValue() klasy Integer.
```

↑
Tworzymy tablicę `ArrayList` obiektów typu `Integer`.
↓

Choć klasa `ArrayList` nie udostępnia metody pozwalającej na bezpośrednie dodawanie liczb typu `int`, to jednak kompilator sam wykonana za nas wszelkie niezbędne konwersje. Innymi słowy, w tablicy `ArrayList` zostanie zapisany obiekt `Integer`, jednak Ty możesz udawać, że zapisujesz i odczytujesz z niej liczby typu `int`. (W tablicy zadeklarowanej jako `ArrayList<Integer>` można przechowywać zarówno wartości typu `int`, jak i obiekty `Integer`).

Nie istniejąca grupa pytań

P: Skoro chcemy stworzyć tablicę przechowującą liczby typu `int`, to czemu nie zadeklarujemy jej jako `ArrayList<int>?`

O: Ponieważ... *nie można* tego zrobić. Pamiętasz zapewne, że według reguł typów ogólnych podczas ich tworzenia można posługiwać się wyłącznie klasami lub interfejsami, a nie typami podstawowymi. Jednak, jak pokazał powyższy przykład, tak naprawdę nie ma to większego znaczenia, gdyż kompilator pozwala umieszczać wartości typu `int` w tablicy `ArrayList<Integer>`. W rzeczywistości, jeśli używamy kompilatora zgodnego z wersją Java 5.0, to taka automatyczna konwersja będzie wykonywana automatycznie, a zatem nic nie może nas powstrzymać przed zapisywaniem w tablicach `ArrayList` wartości typów podstawowych, w przypadku gdy typ tablicy jest typem klasy „opakowującej” danej wartości. A zatem w tablicy `ArrayList<Boolean>` będziemy mogli zapisywać wartości logiczne, a w tablicy `ArrayList<Char>` — znaki.

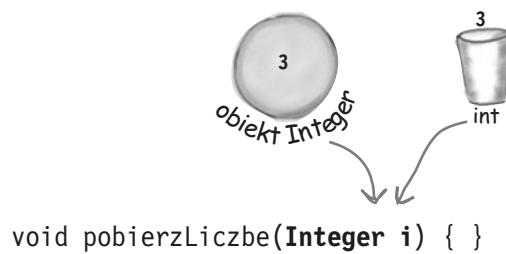
Automatyczna konwersja działa niemal wszędzie

Automatyczna konwersja pomiędzy typami podstawowymi i obiektami pozwala na znacznie więcej niż oczywiste czynności, takie jak zapisywanie wartości typu podstawowego w kolekcji... pozwala także na zamienne stosowanie wartości typu podstawowego i odpowiadającego jej obiektu niemal we wszystkich miejscach, gdzie jest oczekiwana taka wartość lub obiekt. Zastanów się nad tym!

Radość automatycznej konwersji

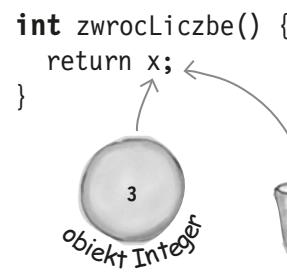
Argumenty metod

Jeśli metoda wymaga obiektu klasy „opakowującej”, to możesz do niej przekazać odwołanie do obiektu odpowiedniego typu bądź wartość typu podstawowego odpowiadającego tej klasie. I na odwrót — jeśli metoda wymaga przekazania wartości typu podstawowego, to w rzeczywistości w jej wywołaniu można przekazać bądź to wartość typu podstawowego, bądź obiekt klasy „opakowującej” odpowiadającej temu typowi podstawowemu.



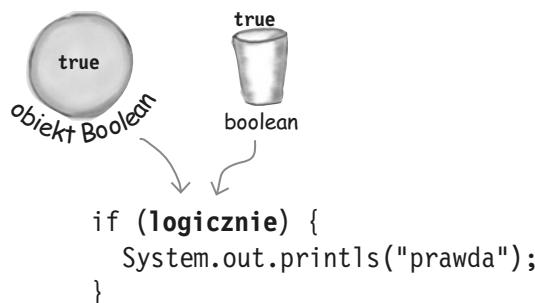
Wartości wynikowe

Jeśli metoda deklaruje wartość wynikową typu podstawowego, to może zwrócić bądź to wartość typu podstawowego, zgodnego z zadeklarowanym, bądź też odwołanie do obiektu klasy „opakowującej” odpowiadającej danemu typowi podstawowemu. I na odwrót, jeśli metoda deklaruje wartość wynikową typu „opakowującego”, to może zwrócić bądź to odwołanie zadeklarowanego typu, bądź wartość typu podstawowego odpowiadającego danej klasie „opakowującej”.



Wyrażenia logiczne

W każdym miejscu, w jakim jest oczekiwane wyrażenie logiczne, można przekazać bądź to takie wyrażenie zwracające wartość logiczną (4>2), bądź wartość logiczną, bądź też obiekt klasy „opakowującej” Boolean.



Operacje na liczbach

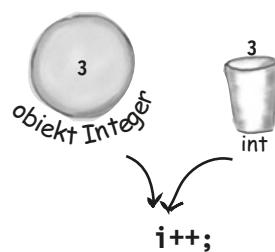
To jest zapewne najciekawsza z tych wszystkich nowych możliwości. Otóż obecnie można używać obiektów klas „opakowujących” jako operandów w operacjach oczekujących wartości typów podstawowych. Oznacza to, że na przykład można użyć operatora inkrementacji wraz z obiektem Integer!

Nie obawiaj się jednak – to są jedynie sztuczki, za którymi stoi kompilator. Nie zmieniono języka i nie udostępnia on możliwości stosowania operatorów wraz z obiekty. Kompilator jedynie skonwertuje obiekt do wartości odpowiedniego typu podstawowego, i dopiero na niej wykona działanie. Bez wątpienia jednak całość wygląda dosyć dziwnie:

```
Integer i = new Integer(42);
i++;
```

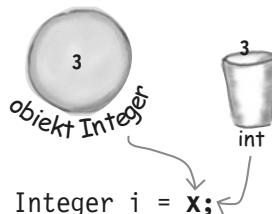
Oznacza to także, że można wykonywać operacje takie jak te pokazane poniżej:

```
Integer j = new Integer(5);
Integer k = j + 3;
```



Przypisania

W zmiennej typu podstawowego bądź zmiennej referencyjnej odpowiedniego typu „opakowującego” można zapisać wartość typu podstawowego lub odpowiadający jej obiekt klasy „opakowującej”. Na przykład w zmiennej typu int można zapisać obiekt klasy Integer i na odwrót – w zmiennej referencyjnej typu Integer można zapisać wartość typu podstawowego int.



Zaostrz ołówek

Czy przedstawiony obok kod zostanie skompilowany? A jeśli uda się go uruchomić, to jakie zwróci wyniki?

Poświęć nieco czasu, aby go przeanalizować i zastanowić się nad nim: przedstawia on pewien aspekt automatycznego konwertowania typów podstawowych i obiektów, o którym jeszcze nie wspominaliśmy.

Będziesz musiał wcielić się w kompilator, aby znaleźć odpowiedź na postawione pytania. (Owszem, zmuszamy Cię do eksperymentowania, ale to dla Twojego własnego dobra).

```
public class MalyTest {
    Integer i;
    int j;

    public static void main (String[] args) {
        MalyBox t = new MalyBox();
        t.go();
    }

    public void go() {
        j=i;
        System.out.println(j);
        System.out.println(i);
    }
}
```

Metody klas reprezentujących typy podstawowe

Chwileczkę! To nie wszystko! Także klasy reprezentujące typy podstawowe dysponują statycznymi metodami pomocniczymi!

Oprócz zachowania charakterystycznego dla standardowych klas, klasy reprezentujące typy podstawowe udostępniają także kilka naprawdę przydatnych metod statycznych. Jednej z nich używaliśmy już we wcześniejszej części książki, była to metoda `Integer.parseInt()`.

Metody przetwarzające pobierająła łańcuch znaków (obiekt `String`) i zwracają wartość typu podstawowego.

Konwersja łańcucha znaków na wartość typu podstawowego jest łatwa:

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");
```

Przetworzenie łańcucha znaków
„2” na liczbę 2 nie przysparza
najmniejszych problemów.

```
boolean b = Boolean.parseBoolean("True");
```

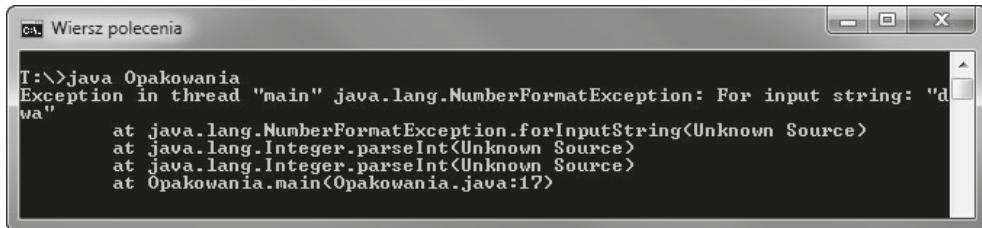
Metoda `parseBoolean()` (nowa,
wprowadzona w wersji 1.5 języka
Java) ignoruje wielkość liter
w przekazanym łańcuchu znaków.

Jeśli jednak spróbujesz postąpić w poniższy sposób:

```
String t = "dwa";
int y = Integer.parseInt(t);
```

Ups! Powyższy kod można skompilować bez
najmniejszego problemu, jednak próba jego
wykonania kończy się tragicznie. Próba
przetworzenia każdego łańcucha znaków, który
nie może być zamieniony na liczbę, spowoduje
zgłoszenie wyjątku `NumberFormatException`.

Podczas próby wykonania zgłoszony zostanie poniższy wyjątek:



The screenshot shows a terminal window titled "Wiersz poleceń". The command entered is "T:\>java Opakowania". The output shows an exception being thrown:

```
T:\>java Opakowania
Exception in thread "main" java.lang.NumberFormatException: For input string: "d
wa"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at Opakowania.main(Opakowania.java:17)
```

Każda metoda lub konstruktor, która przetwarza łańcuchy znaków, może zgłaszać wyjątek `NumberFormatException`. Jest to wyjątek czasu wykonywania programu, zatem nie trzeba go ani deklarować, ani obsługiwać. Lecz mógłbyś chcieć obsługiwać ten wyjątek.

(Zagadnienia związane z wyjątkami zostaną opisane w następnym rozdziale).

A teraz w przeciwnym kierunku... zamiana liczby na łańcuch znaków

Liczbę można zamienić na łańcuch znaków na kilka sposobów. Najprostszy z nich polega na połączeniu liczby z istniejącym obiektem String.

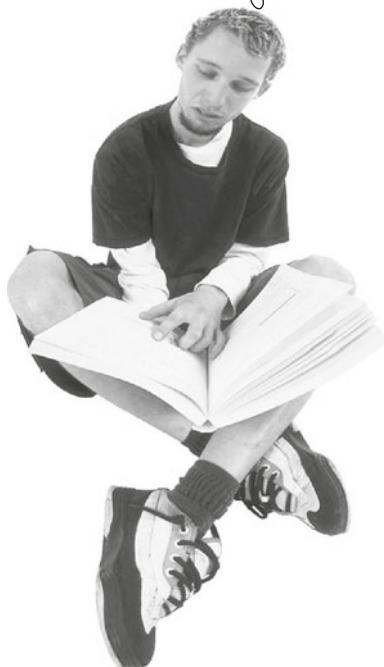
```
double d = 42.5;  
String strDouble = "" + d;  
  
double d = 42.5;  
String strDouble = Double.toString(d);
```

Pamiętaj, że w Javie operator „+” jest przeciążony (zresztą jest to jedyny przypadek przeciążania operatorów, jaki można znaleźć w tym języku) i służy także do łączenia łańcuchów znaków.

Inną metodą jest użycie statycznej metody klasy Double.

Hm... ale jak wyświetlić liczbę w formacie walutowym – z dwoma miejscami po przecinku i symbolem waluty za kwotą, na przykład: 34,99 zł?

A gdzie jest printf?
Czy formatowanie liczb jest częścią biblioteki operacji wejścia-wyjścia?



Formatowanie liczb

W Javie formatowanie liczb nie musi być powiązane z operacjami wejścia-wyjścia. Zastanów się chwilę nad tym. Jednym z najbardziej typowych sposobów prezentowania użytkownikom liczb jest pokazywanie ich na graficznym interfejsie użytkownika. Czyli umieszczasz je w przewijanym obszarze tekstowym albo w tabeli. Gdyby mechanizmy formatowania zostały wbudowane w metodę `print`, nie byłbyś w stanie zapisać ładnie sformatowanej liczby w łańcuchu znaków i następnie wyświetlić w oknie swojej aplikacji. Przed pojawiением się wersji 5.0 języka Java większość mechanizmów formatowani została zaimplementowana w postaci klas umieszczonych w pakiecie `java.text`, jednak teraz wszystko się zmieniło i w tym wydaniu książki nawet do nich nie zajrzymy.

W wersji 5.0 Javy twórcy języka dodali znacznie bardziej potężne i elastyczne możliwości formatowania, udostępnione jako klasa `Formatter` umieszczona w pakiecie `java.util`. Jednak nie musisz samemu tworzyć obiektów i wywoływać metod tej klasy — gdyż specjalne metody, ułatwiające korzystanie z mechanizmów formatowania, zostały dodane do wybranych klas i metod związanych z operacjami wejścia-wyjścia (w tym także do metody `printf()`) oraz do klasy `String`. A zatem wszystko sprowadza się aktualnie do wywołania statycznej metody `String.format()` i przekazania do niej wartości, którą chcemy sformatować, oraz instrukcji, jak należy to zrobić.

Oczywiście, trzeba wiedzieć, jak określić instrukcje dotyczące sposobu formatowania, a to może wymagać pewnego wysiłku, chyba że znamy funkcję `printf()` stosowaną w językach C i C++. Na szczęście, nawet jeśli jej *nie* znasz, możesz postępować zgodnie z wytycznymi formatowania podstawowych rodzajów informacji (przedstawionymi w dalszej części tego rozdziału). Niemniej jednak, jeśli chcesz wiedzieć, jak formatować dowolne dane i uzyskiwać *dowolne* zamierzane efekty, to zapewne *będziesz* chciał nauczyć się instrukcji sterujących formatowaniem.

Zaczniemy od prostego przykładu i przyjrzymy się, jak to wszystko działa. (Uwaga: zagadnieniem formatowania zajmiemy się jeszcze raz, w rozdziale poświęconym operacjom wejścia-wyjścia).

Formatowanie liczb z użyciem spacji

```
public class TestFormatowania {  
    public static void main (String[] args) {  
        String s = String.format("%, d", 1000000000);  
        System.out.println(s);  
    }  
}
```

1 000 000 000

Liczba, która chcemy sformatować (chcemy, by została ona podzielona na grupy 3 cyfr oddzielone od siebie spacją).

Instrukcje określające, jak należy sformatować drugi argument wywołania metody (którym w naszym przypadku jest liczba całkowita). Pamiętaj, że ta metoda ma tylko dwa argumenty — pierwszy przecinek jest umieszczony WEWNĄTRZ literatu łańcuchowego, a zatem nie służy do oddzielenia od siebie argumentów wywołania metody `format()`.

↑ Uzyskałeś liczbę z dodanymi spacjami.

Formatowanie rozmontowane na czynniki pierwsze

W najprostszym możliwym ujęciu formatowanie składa się z dwóch podstawowych elementów (w rzeczywistości jest ich więcej, jednak zaczniemy tylko od dwóch, by zbytnio nie komplikować sprawy).

1 Instrukcje formatowania

Specjalnych specyfikatorów formatu używasz, by opisać, w jaki sposób należy sformatować argument.

Uwaga: Jeśli potrafisz się posługiwać funkcją printf() stosowaną w językach C i C++, to najprawdopodobniej możesz pominać kilka następnych stron. Jednak w przeciwnym razie przeczytaj je uważnie!

2 Formatowany argument

Choć argumentów tych może być więcej, to zaczniemy od jednego. Nie może to być argument *dowolnego* typu — musi to być coś, co można sformatować przy użyciu określonych wcześniej instrukcji formatowania. Na przykład: jeśli w instrukcjach określmy, że będzie formatowana *liczba zmiennoprzecinkowa*, to nie będziemy mogli przekazać do metody obiektu klasy Pie ani nawet łańcucha znaków mającego postać liczby zmiennoprzecinkowej.

Zrób to... ... z tą wartością.

```
①           ②
format("%, d", 1000000000);
```

Użyj tych instrukcji... by sformatować ten argument.

Co tak naprawdę nakazują te instrukcje?

„Weź drugi argument wywołania metody i sformatuj go jako liczbę całkowitą (**d**) z dodatkowymi **spacjami** oddzielającymi grupy cyfr.”

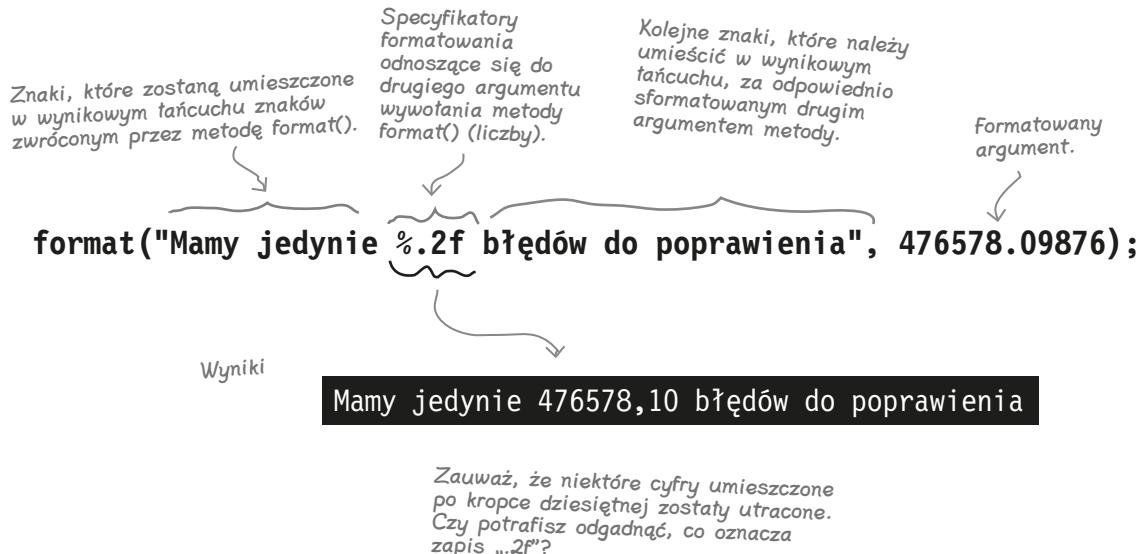
Jak to zapisano?

Dokładny opis znaczenia instrukcji formatowania "%, d" można znaleźć na następnej stronie, tu jednak podamy nieco informacji, aby od czegoś zacząć. Za każdym razem, gdy w instrukcji formatowania (stanowiącej pierwszy argument wywołania metody format()) zobaczysz znak procentu (%), wyobraź sobie, że reprezentuje on zmienną, a zmienna ta jest kolejnym argumentem wywołania metody. Kolejne znaki umieszczone za procentem określają instrukcje formatowania odnoszące się do tego argumentu.

Metoda format()

Znak procentu (%) oznacza: „tu wstaw argument” (i sformatuj go zgodnie z podanymi instrukcjami)

Pierwszy argument wywołania metody `format()` jest nazywany łańcuchem formatującym; może on zawierać znaki, które chcemy umieścić w niezmienionej postaci w wynikowym, sformatowanym łańcuchu znaków. Jeśli jednak w łańcuchu formatującym zobaczymy znak procentu (%), to powinniśmy sobie wyobrazić, że reprezentuje on zmienną przekazaną jako kolejny argument wywołania metody.



Znak procentu (%) informuje mechanizm formatujący, że w danym miejscu należy umieścić inny argument wywołania metody (w tym przypadku będzie to drugi argument, czyli liczba) i sformatować go przy wykorzystaniu specyfikatora o postaci ".2f", podanego za znakiem procentu. Dalsza część łańcucha formatującego, czyli słowa „ błędów do poprawienia”, zostanie dodana do wynikowego łańcucha znaków w niezmienionej postaci.

Dodawanie spacji

```
format("Mamy jedynie %,.2f błędów do poprawienia", 476578.09876);
```

Mamy jedynie 476 578,10 błędów do poprawienia

Zmieniając instrukcję formatowania z „%.2f” na „%,.2f”, udało się nam umieścić spację w wynikowym łańcuchu znaków.



Ale skąd ta metoda wie, gdzie kończą się instrukcje, a zaczynają zwyczajne znaki? Dlaczego nie wyświetla litery „f” z instrukcji formatowania „%.2f”? Skąd ona wie, że .2f to instrukcja formatowania, a NIE część łańcucha znaków, którą należy zostawić w niezmienionej postaci?

Łańcuch formatowania ma swoją własną, prostą składnię

Oczywiście, za znakiem procentu (%) w łańcuchu formatującym nie można umieszczać dowolnych znaków. Składnia określająca, co można tam zapisać, jest sprecyzowana i opisuje, jak należy sformatować argument, który zostanie umieszczony w danym miejscu wynikowego (sformatowanego) łańcucha znaków.

Poznałeś już dwa przykłady:

„%, d” oznacza „wstaw spację i sformatuj argument jako liczbę całkowitą (w systemie dziesiętnym)”.

Oraz:

„%.2f” oznacza „sformatuj wartość jako liczbę zmienoprzecinkową i wyświetl dwie cyfry po kropce dziesiętnej”.

Oraz:

„,.2f” oznacza „wstaw spację i sformatuj wartość jako liczbę zmienoprzecinkową z dwoma miejscami dziesiętnymi”.

Jednak oto prawdziwe pytanie, jakie należy sobie zadać: „Co należy zapisać po znaku procentu, by zmusić metodę do sformatowania wartości w taki sposób, jak bym sobie tego życzył?”. Odpowiedź na to pytanie wiąże się z poznaniem symboli (takich jak „d” oznaczające liczby całkowite zapisane w systemie dziesiętnym oraz „f” reprezentujące liczby zmienoprzecinkowe) oraz kolejności, w jakiej należy umieszczać poszczególne instrukcje za znakiem procentu. Na przykład: jeśli umieścimy przecinek za symbolem „d”, czyli zmienimy łańcuch formatujący z „%,d” na „%d,”, to nie zadziała on zgodnie z naszymi oczekiwaniami!

A może jednak zadziała? Jak sądzisz, jaki będzie łańcuch znaków zwrócony przez poniższe wywołanie:

```
format("Mamy jedynie %.2f, błędów do poprawienia", 476578.09876);
```

(Odpowiedź znajdziesz na następnej stronie).

Specyfikator formatu

Wszystko, co zostanie umieszczone za znakiem procentu aż do określenia typu (np.: "d" lub "f", łącznie z nim), jest częścią instrukcji formatowania. Mechanizm formatujący uznaje jednak, że wszystkie znaki umieszczone za określeniem typu (aż do końca łańcucha lub do miejsca wystąpienia kolejnego znaku procentu) mają się znaleźć w wynikowym łańcuchu znaków w niezmienionej postaci. Hmm... czy to w ogóle jest możliwe? Czy można przekazać więcej niż jeden argument do sformatowania? Wstrzymajmy się na razie z odpowiedzią na to pytanie, zajmiemy się tym zagadnieniem za kilka minut. Na razie przyjrzyjmy się składni specyfikatorów formatu — czyli znakom umieszczanym za znakiem procentu i opisującym, jak powinien zostać sformatowany argument.

Specyfikator formatu może składać się aż z pięciu różnych elementów (nie uwzględniając znaku procentu). Wszystkie elementy, które na poniższym opisie zostały zapisane w nawiasach kwadratowych, należy uznać za opcjonalne. Oznacza to, że wymaganymi elementami specyfikatora formatu są: znak procentu oraz określenie typu. Jednak także kolejność, w jakiej te elementy są zapisywane, jest ściśle określona, dlatego też wszystkie elementy specyfikatora formatu, których zdecydujemy się użyć, będą musiały być zapisane w przedstawionej kolejności.

`%[numer argumentu] [flagi] [szerokość] [.precyzja] typ`

Tym zajmiemy się za chwilę... tu można określić, KTÓRY argument chcemy sformatować (jeśli jest ich więcej). (Na razie się tym nie przejmuj).

Tu można podawać specjalne opcje formatowania, takie jak dodawanie spacji, umieszczać liczb ujemnych w nawiasach lub wyrównywanie liczb do lewej.

W tym miejscu możemy określić MÍNIMALNA ilość znaków, jaka zostanie użyta do przedstawienia liczby. To ilość "minimalna", a nie SUMARYCZNA. Jeśli ilość znaków koniecznych do przedstawienia liczby jest większa od tej wartości, to liczba i tak zostanie przedstawiona w całości; jednak w przeciwnym razie zostanie ona poprzedzona zerami.

To już znasz... w ten sposób można określić precyzję — czyli ilość liczb po kropce dziesiętnej. Nie zapomij poprzedzić cyfry znakiem kropki (.).

Określenie typu jest obowiązkowe (wyjaśniamy to na następnej stronie) i zazwyczaj będzie przyjmować postać litery "d" (dla liczb całkowitych zapisanych w systemie dziesiętnym) lub "f" (dla liczb zmienoprzecinkowych).

`%[numer argumentu] [flagi] [szerokość] [.precyzja] typ`

`format("%,6.1f", 42.000);`

W tym łańcuchu formatującym zabrakło określenia „numeru argumentu”, jednak wszystkie pozostałe dostępne elementy są na swoich miejscach.

Jednym wymaganym specyfikatorem jest modyfikator TYPU

Choć określenie typu jest jednym wymaganym specyfikatorem, to jednak musisz pamiętać, że jeśli wykorzystasz jakiekolwiek inne elementy specyfikatora formatu, to typ musi zostać podany jako ostatni! Dostępnych jest kilkanaście różnych modyfikatorów typu (nie licząc tych związanych z datami i czasem, gdyż one są zaliczane do osobnej grupy), jednak zapewne najczęściej stosowanymi są te: %d (liczby całkowite, w systemie dziesiętnym) oraz %f (liczby zmiennoprzecinkowe). Dodatkowo modyfikator typu %f będzie łączony z określeniem precyzji, pozwalającym podać ilość cyfr po kropce dziesiętnej.

TYP jest obowiązkowy, wszelkie pozostałe elementy specyfikatora formatu są opcjonalne.

%d Liczba całkowita, w systemie dziesiętnym

```
format("%d", 42);
```

42

Przekazanie wartości 42.35 nie dałoby zamierzonych efektów! To tak samo jakbyś próbował bezpośrednio zapisać wartość typu double w zmiennej typu int.

Argument musi być typu zgodnego z typem int, a to oznacza, że mogą to być wartości byte, short, int oraz char (albo obiekty odpowiadających im klas „opakowujących”).

%f Liczba zmiennoprzecinkowa

```
format("%.3f", 42.000000);
```

42.000

W tym przykładzie łączymy modyfikator typu „f” z określeniem precyzji „3”, dzięki czemu wynikowa liczba będzie miała trzy cyfry po kropce dziesiętnej.

Argument musi być wartością zmiennoprzecinkową, czyli wchodzą w grę jedynie typy float oraz double (lub odpowiadające im klasy „opakowujące”) oraz coś, co jest określone jako BigInteger (i czym nie będziemy się zajmować w tej książce).

%x Liczba zapisana w systemie szesnastkowym

```
format("%x", 42);
```

2a

Argumentem musi być wartość typu byte, short, int, long (bądź obiekt odpowiadającej im klasy „opakowującej”) lub BigInteger.

%c Znak

```
format("%c", 42);
```

*

Liczba 42 reprezentuje znak „”

Argumentem musi być wartość typu byte, short lub int (bądź obiekt odpowiadającej im klasy „opakowującej”).

W instrukcjach formatowania koniecznie trzeba podać modyfikator typu, a jeśli oprócz niego podamy coś jeszcze, to modyfikator typu musi zostać zapisany jako ostatni. W przeważającej większości przypadków będziesz formatował liczby, wykorzystując typy „d” (by wyświetlać liczby całkowite w systemie dziesiętnym) oraz „f” (by wyświetlać liczby zmiennoprzecinkowe).

Co się stanie, jeśli użyję więcej niż jednego argumentu?

Wyobraź sobie, że chciałbyś uzyskać łańcuch znaków o następującej postaci:

„Uzyskałeś wynik **20,456,654** punkty z **100,576,890.24** możliwych.”

Jednak wartości te mają pochodzić ze zmiennych. Co zrobić w takim przypadku? Otóż w wywołaniu metody `format()` za łańcuchem formatującym wystarczy podać *dwa* argumenty. Oznacza to, że w wywołaniu tej metody znajdą się w sumie trzy argumenty. Oprócz tego wewnątrz pierwszego argumentu (łańcucha znaków określającego format) trzeba będzie podać dwa specyfikatory formatu (ciagi znaków zaczynające się od znaku procentu). Pierwszy specyfikator formatu określi postać i umieści w wynikowym łańcuchu znaków wartość drugiego argumentu wywołania metody `format()`, natomiast drugi specyfikator — wartość trzeciego argumentu. Innymi słowy, wartości są wstawiane do łańcucha formatującego w takiej samej kolejności, w jakiej argumenty zostały umieszczone w wywołaniu metody `format()`.

```
int jeden = 20456654;  
double dwa = 100567890.248907;  
String s = String.format("Uzyskałeś wynik %,d punkty z %,.2f możliwych", jeden, dwa);
```

Uzyskałeś wynik 20 456 654 punkty z 100 576 890,24 możliwych.

Do obu wartości dodaliśmy spacje,
a w drugiej z nich ograniczyliśmy
także ilość wyświetlanych miejsc
dziesiętnych do dwóch.

Jeśli użyjesz więcej niż jednego argumentu, to będą one umieszczane w wynikowym łańcuchu znaków w takiej samej kolejności, w jakiej zostały podane w wywołaniu metody `format()`.

Jak się przekonasz, gdy zajmiemy się zagadnieniami formatowania dat, może się zdarzyć, że będziesz chciał podać kilka różnych specyfikatorów formatu odnoszących się do tego samego argumentu. Zapewne trudno będzie Ci to sobie wyobrazić aż do czasu, gdy poznasz formatowanie *dat* (a nie jedynie *liczby*, którymi zajmujemy się do tej pory). Na razie wystarczy, byś wiedział, że już wkrótce przekonasz się, w jaki sposób można precyzyjnie określić, który specyfikator formatu odnosi się do którego argumentu.

Nie istnieją
głupie pytania

P: Hm... tu dzieje się coś NAPRAWDĘ dziwnego. Jak dużo argumentów można przekazać w wywołaniu metody `format()`? Czyli ile przeciążonych wersji tej metody zdefiniowano w klasie `String`? Innymi słowy, co się stanie, kiedy spróbuję przekazać w wywołaniu metody `String` na przykład dziesięć argumentów i umieścić je wszystkie w łańcuchu wynikowym?

O: Niezła sztuczka. Owszem, w tym jest coś dziwnego (albo przynajmniej nowego i innego), ale nie — nie ma całej grupy przeciążonych metod `format()` umożliwiających podanie różnych ilości argumentów. W celu udostępnienia tych nowych możliwości formatowania (przypominających metodę `printf()`) język Java potrzebował zupełnie nowej cechy, tak zwanej *zmiennej listy argumentów* (określonej także skrótnie jako *varargs*¹). Zajmiemy się nią w dodatku, gdyż z wyjątkiem formatowania łańcuchów znaków, możliwość ta nie będzie zbyt często używana w dobrze zaprojektowanych programach.

¹ Od angielskich słów: variable argument list.

Tak dużo z myślą o liczbach, a co z datami?

Wyobraź sobie, że chciałbyś uzyskaćłańcuch znaków o postaci „Niedziela, 28 paz 2004”.

Mówisz, że nie ma w nim nic szczególnego? Cóż, wyobraź sobie na początku, że masz zmienną typu Date — w Javie jest to klasa reprezentująca znacznik czasu — i chciałbyś przekazać ten obiekt (bo to już nie jest liczba) do metody formatującej.

Podstawową różnicą pomiędzy formatowaniem liczb i dat jest to, że w przypadku dat modyfikatory typu składają się nie z jednego (np. „d” lub „f”), lecz z dwóch znaków, z których pierwszym jest litera „t”. Poniższe przykłady powinny doskonale wyjaśnić Ci, jak to działa:

Pełna data i godzina **%tc**

```
String.format("%tc", new Date());
```

N lis 28 20:34:45 CEST 2004

Tylko godzina **%tr**

```
String.format("%tr", new Date());
```

11:19:25 PM

Dzień tygodnia, miesiąc i dzień **%tA %tB %td**

Nie ma jednego specyfikatora formatu, który wyświetliłby datę dokładnie w takiej postaci, w jakiej byśmy chcieli, dlatego też musimy połączyć trzy odrębne specyfikatory prezentujące: dzień tygodnia (%tA), miesiąc (%tB) oraz numer dnia w miesiącu (%td).

```
Date dzis = new Date();
String.format("%tA, %tB %td", dzis, dzis, dzis);
```

↑
Ten przecinek nie jest częścią specyfikatora formatu... to po prostu znak, który chcemy wyświetlić po pierwszej sformatowanej wartości.

niedziela, listopad 28

Jednak to oznacza, że w wywołaniu metody format() musimy trzy razy przekazać ten sam obiekt Date — po razie dla każdego specyfikatora. Innymi stowry, specyfikator %tA zwróci nam jedynie nazwę dnia tygodnia, a później musimy ponownie przekazać ten sam obiekt daty, by uzyskać nazwę miesiąca i numer dnia w miesiącu.

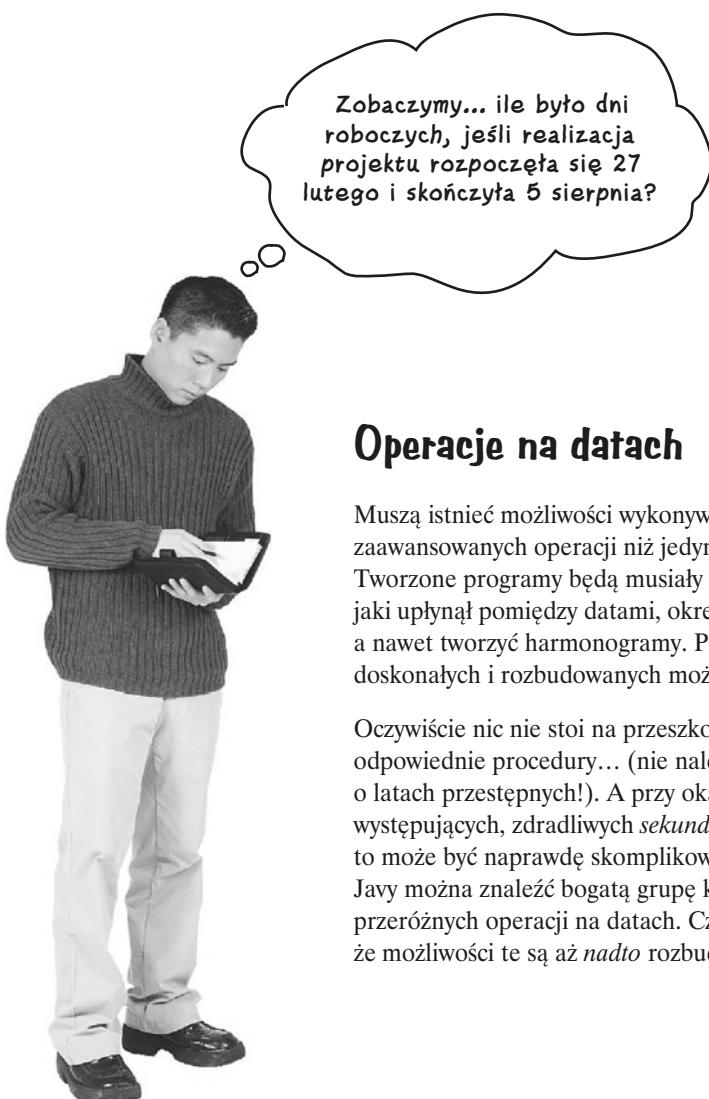
To samo co powyżej, lecz bez powielania argumentów

```
Date dzis = new Date();
String.format("%tA, %<td %<td", dzis);
```

Możesz to sobie wyobrazić jako wywołanie trzech różnych metod pobierających na jednym obiekcie Date w celu pobrania z niego trzech różnych informacji.

%tA %<tB %<td

Nawias kątowy „<” jest kolejną flagą, jaką można stosować w specyfikatorach formatu. Nakazuje ona ponowne wykorzystanie ostatniego argumentu. A zatem pozwala uniknąć niepotrzebnego powielania argumentów i zapewnia możliwość sformatowania tego samego argumentu na kilka różnych sposobów.



Operacje na danych

Muszą istnieć możliwości wykonywania na danych bardziej zaawansowanych operacji niż jedynie określenie bieżącej daty. Tworzone programy będą musiały zmieniać daty, określać czas, jaki upłynął pomiędzy datami, określać priorytety harmonogramów, a nawet tworzyć harmonogramy. Potrzebujemy naprawdę doskonałych i rozbudowanych możliwości manipulowania danymi.

Oczywiście nic nie stoi na przeszkodzie, byśmy sami napisali odpowiednie procedury... (nie należy przy tym zapominać o latach przestępnych!). A przy okazji o tych sporadycznie występujących, zdradliwych *sekundach* przestępnych. O rany... to może być naprawdę skomplikowane. Na szczęście w bibliotece Javy można znaleźć bogatą grupę klas ułatwiających wykonywanie przeróżnych operacji na danych. Czasami można odnieść wrażenie, że możliwości te są aż *nadto* rozbudowane...

Poruszanie się wzdłuż osi czasu

Załóżmy, że w Twojej firmie pracuje się od poniedziałku do piątku. Przydzielono Ci zadanie określenia dat ostatniego dnia roboczego każdego miesiąca w tym roku.

Wydaje się, że klasa `java.util.Date`... nieco się przedawniła

Wcześniej w tym rozdziale skorzystaliśmy z klasy `java.util.Date`, by określić aktualną datę. Można by zatem przypuszczać, że będzie ona także stanowić wygodny punkt startowy dla poszukiwań mechanizmów manipulowania datami. Kiedy jednak zajrzymy do biblioteki Javy, okaże się, że większość metod klasy `Date` została uznana za przedawnione!

Klasa `Date` wciąż doskonale nadaje się do pobierania „znacznika czasu” — czyli obiektu reprezentującego bieżącą datę i czas. Możesz jej zatem z powodzeniem używać, kiedy chcesz zadać pytanie: „Powiesz mi, która jest TERAZ godzina?”.

Na szczęście biblioteka Javy zaleca korzystanie z klasy `java.util.Calendar` zamiast z klasy `Date`, a zatem przyjrzyjmy się jej:

Używaj klas `java.util.Caledar` do wykonywania operacji na datach

Projektanci tworzący interfejs programistyczny do obsługi kalendarzy i dat chcieli myśleć globalnie, i to dosłownie. Podstawowa idea polega na tym, że jeśli chcemy pracować na datach, musimy poprosić o kalendarz (przy użyciu statycznej metody klasy `Calendar`, którą poznasz na następnej stronie), a JVM zwróci nam obiekt konkretnej klasy potomnej klasy `Calendar`. (`Calendar` jest klasą abstrakcyjną, a zatem zawsze będziemy operować na konkretnej klasie potomnej).

Znacznie ciekawsze jest jednak to, że rodzaj kalendarza, jaki uzyskamy, będzie odpowiadał aktualnym ustawieniom lokalnym. W większości krajów na świecie używa się kalendarza gregoriańskiego, jeśli jednak znajdujesz się w jednym z niewielu miejsc, gdzie używany jest inny kalendarz, taki jak buddyjski, islamski bądź japoński, to biblioteka Javy będzie w stanie Ci go udostępnić.

Standardowa biblioteka Javy udostępnia klasę `java.util.GregorianCalendar`, a zatem to właśnie jej będziemy tu używać. Jednak w przeważającej większości przypadków nie będziesz musiał zwracać sobie głowy rodzajem używanej klasy kalendarza — będziesz mógł się skoncentrować na jej metodach.

Aby pobrać znacznik czasu odpowiadający „bieżącej chwili”, użyj klasy `Date`.

We wszystkich innych przypadkach używaj klasy `Calendar`.

Pobieranie obiektu klasy potomnej klasy Calendar

Niby w jaki sposób możemy utworzyć i pobrać obiekt klasy abstrakcyjnej?

Cóż, tego oczywiście nie można zrobić. Poniższy fragment kod nie zadziała:

To NIE zadziała:

```
Calendar kal = new Calendar();
```

Kompilator na to nie pozwoli!

Trzeba użyć statycznej metody getInstance():

```
Calendar kal = Calendar.getInstance();
```

Obecnie ta składnia powinna już wyglądać znajomo — wywołujemy tu metodę statyczną.

Chwileczkę. Skoro nie można utworzyć obiektu klasy Calendar, to właściwie co zapisujemy w zmiennej typu Calendar?



Nie możesz pobrać obiektu klasy Calendar, jednak możesz utworzyć obiekt którejś z jej klas potomnych.

Oczywiście nie możesz utworzyć obiektu klasy Calendar — jest to bowiem klasa abstrakcyjna. Nic jednak nie stoi na przeszkodzie, by wywoływać *statyczne* metody za pośrednictwem samej *klasy*, bez odwoływanego się do jej obiektów. A zatem wystarczy, że wywołasz statyczną metodę `getInstance()` klasy `Calendar`, a uzyskasz... obiekt jednej z jej konkretnych klas potomnych. Oznacza to, że można go polimorficznie zapisać w zmiennej typu `Calendar`, oraz że na mocy swojego kontraktu będzie on udostępniał metody klasy `Calendar`.

W większości miejsc na świecie, oraz domyślnie w większości wersji języka Java, wywołanie metody `getInstance()` klasy `Calendar` zwróci obiekt klasy `java.util.GregorianCalendar`.

Stosowanie obiektów Calendar

Aby móc posługiwać się obiekta Calendar, będziesz musiał zrozumieć kilka zagadnień.

- **Pola przechowują stan** — Obiekt Calendar posiada wiele pól (ang. *field*) używanych do przechowywania różnych aspektów właściwego stanu obiektu — czyli reprezentowanej przez niego daty i czasu. Na przykład możesz określić lub odczytać z kalendarza określony przez niego rok oraz miesiąc.
- **Daty i godziny można inkrementować** — Klasa Calendar udostępnia metody pozwalające dodawać i odejmować wartości od różnych pól, na przykład można wykonać operacje: „dodaj dwa do numeru miesiąca” lub „odejmij trzy od roku”.
- **Daty i godziny mogą być reprezentowane jako milisekundy** — Klasa Calendar pozwala zapisywać daty w formie milisekund. (A konkretnie rzecz biorąc, jako liczbę milisekund, jaka upłynęła od 1 stycznia 1970 roku). Dzięki temu zyskujemy możliwość wykonywania precyzyjnych obliczeń typu wyliczenia „czasu, jaki upłynął pomiędzy dwiema datami” lub „dodania do daty 63 godzin, 15 minut i 23 sekund”.

Przykłady operacji na obiekcie Calendar:

```
Calendar c = Calendar.getInstance();
c.set(2004, 0, 7, 15, 40);           ← Ustawiamy datę na 7 stycznia 2004
                                              roku, na godzinę 15:40.

long d1 = c.getTimeInMillis();        ← Konwertujemy datę na stałą, dobrą
                                              liczbę określającą liczbę milisekund.

d1 += 1000 * 60 * 60;                ← Dodajemy godzinę wyrażoną jako liczba milisekund,
                                              po czym aktualizujemy czas. (Zauważ, że operator
                                              „+=” oznacza to samo, co d1 = d1 + ...).

c.setTimeInMillis(d1);             ← Przesuwamy” datę o 35 dni. Ta operacja
                                              powoduje przesunięcie daty o 35 dni, JEDNAK
                                              BEZ zmiany miesiąca!

System.out.println("Nowa godzina to: " + c.get(c.HOUR_OF_DAY));

c.add(c.DATE, 35);                 ← Dodajemy do daty 35 dni, co powinno
                                              przesunąć ją do lutego.

System.out.println("Po dodaniu 35 dni: " + c.getTime());
c.roll(c.DATE, 35);               ← Przesuwamy” datę o 35 dni. Ta operacja
                                              powoduje przesunięcie daty o 35 dni, JEDNAK
                                              BEZ zmiany miesiąca!

System.out.println("Po przesunięciu daty o 35 dni: " + c.getTime());

c.set(c.DATE, 1);                  ← W tym przypadku nie inkrementujemy
                                              daty — po prostu ją ustawiamy.

System.out.println("Ustawiono dzień na 1.: " + c.getTime());
```

```
T:\>>java Kalendarze
Nowa godzina to: 16
Po dodaniu 35 dni: Wed Feb 11 16:40:50 CET 2004
Po przesunięciu daty o 35 dni: Tue Feb 17 16:40:50 CET 2004
Ustawiono dzień na 1.: Sun Feb 01 16:40:50 CET 2004
```

Te wyniki potwierdzają prawidłowe działanie pobierania czasu w milisekundach, dodawania dni, przesuwania oraz ustawiania daty.

Wybrane możliwości API klasy Calendar

Właśnie poznaliśmy sposoby korzystania z kilku pól i metod klasy `Calendar`. Jej możliwości są bardzo duże, dlatego też pokazujemy tu jedynie kilka wybranych metod i pól, z których będziesz najczęściej korzystał. Kiedy je opanujesz, nie powinieneś mieć problemu z używaniem pozostałych zgodnie ze swoją wolą.

Kluczowe metody klasy Calendar

`add(int pole, int liczba)`

Dodaje lub odejmuje czas od wybranego pola kalendarza.

`get(int pole)`

Zwraca wartość wybranego pola kalendarza.

`getInstance()`

Zwraca obiekt `Calendar`, przy okazji można określić ustawienia lokalne.

`getTimeInMillis()`

Zwraca czas kalendarza wyrażony jako liczba milisekund (w postaci wartości typu `long`).

`roll(int pole, boolean wPrzod)`

Dodaje lub odejmuje jednostkę czasu od wybranego pola, bez zmieniania wartości „większych” pól kalendarza.

`set(int pole, int wartosc)`

Ustawia wartość wskazanego pola kalendarza.

`set(rok, miesiąc, dzień, godzina, minuta)`

Często stosowana metoda określająca pełną datę i godzinę.

`setTimeInMillis()`

Ustawia czas kalendarza na podstawie czasu wyrażającego liczbę milisekund (liczba typu `long`).

// wiele innych...

Kluczowe pola klasy Calendar

`DATE / DAY_OF_MONTH`

Pozwala odczytać lub ustawić dzień miesiąca

`HOUR / HOUR_OF_DAY`

Pozwala odczytać lub ustawić godzinę (w zapisie 12- lub 24-godzinnym).

`MILLISECOND`

Pozwala odczytać lub ustawić czas w milisekundach.

`MINUTE`

Pozwala odczytać lub ustawić minutę.

`MONTH`

Pozwala odczytać lub ustawić miesiąc.

`YEAR`

Pozwala odczytać lub ustawić rok.

`ZONE_OFFSET`

Pozwala odczytać lub ustawić przesunięcie w stosunku do czasu GMT (wyrażone w milisekundach).

// wiele innych...

Jeszcze więcej elementów statycznych!... Import statyczny

Zaczynasz spotkanie z Java od wersji 5.0? To prawdziwe (i jednocześnie wątpliwe) błogosławieństwo. Niektórzy ubóstwiają to rozwiązanie, a inni go nienawidzą. Import statyczny wprowadzono wyłącznie po to, by zaoszczędzić programistom trochę pisania. Jeśli nienawidzisz niepotrzebnie stukać w klawiaturę, to rozwiązanie to może Ci się spodobać. Jednak import statyczny ma pewną wadę — jeśli nie będzie stosowany uważnie, może sprawić, że kod będzie znacznie trudniejszy do analizy.

Pomysł polega na tym, że za każdym razem, gdy używasz statycznej klasy, zmiennej lub typu wyliczeniowego, możesz je zimportować i oszczędzić sobie trochę pisania.

Kod zapisany w tradycyjny sposób:

```
import java.lang.Math;
class BezStatycznegoImportu {
    public static void main(String[] args) {
        System.out.println("Pierwiastek z dwóch to: " + Math.sqrt(2.0));
        System.out.println("Tangens: " + Math.tan(60));
    }
}
```

Ten sam kod z wykorzystaniem importów statycznych:

```
import static java.lang.System.out;
import static java.lang.Math.*;

class ZUzyciemStatycznegoImportu {
    public static void main(String[] args) {
        out.println("Pierwiastek z dwóch to: " + sqrt(2.0));
        out.println("Tangens: " + tan(60));
    }
}
```

Import statyczny w działaniu

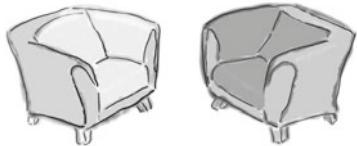
**Stosować uważnie:
używanie importu
statycznego może
sprawić, że kod będzie
trudny do zrozumienia.**

CELNE SPOSTRZEŻENIA

- Jeśli statycznej składowej chcesz używać jedynie kilka razy, to uważamy, że powinieneś unikać stosowania importów statycznych, poprawiając tym samym łatwość zrozumienia kodu.
- Jeśli wiesz, że składowych statycznych będziesz używał bardzo często (na przykład w razie wykonywania obliczeń matematycznych), to można sądzić, że zastosowanie importu statycznego może być uzasadnione.
- Zauważ, że w deklaracji importu statycznego można skorzystać ze znaków wieloznacznego (*).
- Poważnym problemem związanym z importem statycznym jest to, że stosunkowo łatwo można doprowadzić do powstania konfliktów nazw. Na przykład, jeśli dysponujesz dwiema klasami, z których każda udostępnia metodę dodaj(), to w jaki sposób zarówno Ty, jak i kompilator macie się zorientować, której z nich należy użyć?

Składowa statyczna i składowa

Pogawędki przy kominku



Składowa

W ogóle nie rozumiem, po co to robimy. Wszyscy wiedzą, że składowe statyczne są wykorzystywane wyłącznie do tworzenia stałych. Poza tym, jak wiele jest takich składowych statycznych? Myślę, że w całym API jest ich pewnie nie więcej niż cztery. A w ogóle, to jest mało prawdopodobne, aby ktoś ich używał.

Pełno składowych statycznych! Pewnie... możesz sobie powtarzać. No dobrze, może i jest ich trochę w bibliotece Swing, ale wszyscy wiedzą, że Swing to szczególny przypadek.

No dobrze, ale podaj mi choć jeden przykład przydatnej i faktycznie wykorzystywanej składowej statycznej zdefiniowanej w klasie, która nie jest związana z graficznym interfejsem użytkownika.

No dobrze, ale to kolejny przypadek szczególny. A poza tym nikt nie korzysta z tej składowej, chyba że podczas testowania programów.

**Temat dzisiejszej pogawędki:
Składowa wyśmiewająca się ze składowej statycznej.**

Składowa statyczna

Naprawdę powinnaś się douczyć. Kiedy ostatni raz zaglądałaś do dokumentacji Java API? W API jest cała masa składowych statycznych! Istnieje nawet specjalna klasa przeznaczona do przechowywania stałych. Istnieją też klasy, w których jest pełno składowych statycznych, na przykład `SwingConstants`.

Swing może i jest szczególnym przypadkiem, ale za to ważnym! A co powiesz o klasie `Color`? Wyobrażasz sobie, jak kłopotliwe byłoby zapamiętywanie tych wszystkich wartości RGB odpowiadających standardowym kolorom? Ale na szczęście w klasie `Color` zostały zdefiniowane stałe dla podstawowych kolorów, takich jak niebieski, czerwony, biały i tak dalej. To bardzo wygodne.

Może na początek `System.out`? To „out” w `System.out` to składowa statyczna klasy `System`. Sama nie tworzysz nowego obiektu klasy `System`, zatem prosisz ją o zwrócenie wartości jej składowej `out`.

A co, może testowanie nie jest ważne?

A poza tym jest jeszcze jedna sprawa, która prawdopodobnie nigdy ci nie wpadła do tej ograniczonej główka. Musisz się z tym pogodzić, ale składowe statyczne są bardziej efektywne od zwyczajnych składowych. Jedna na klasę, a nie po jednej w każdym obiekcie. Oszczędności pamięci mogą być ogromne!

Składowa

Pewnie, ale czy nie zapominasz o czymś?

Składowe statyczne są antyobiektowe! Dlaczego po prostu nie zrobić gigantycznego kroku wstecz i wrócić do programowania proceduralnego.

Jesteś jak zwyczajna zmienna globalna, a każdy programista wart swojego komputera PDA wie, że zazwyczaj zmienne globalne są złe.

Może i istniejesz w klasie, ale nie można tego nazwać programowaniem zorientowanym *klasowo*. To po prostu głupie. Stanowisz wymierający gatunek. Coś, co istnieje tylko po to, by pomóc starszym programistom korzystać z Javy.

No dobrze, od czasu do czasu użycie składowej statycznej ma sens, ale coś ci powiem — zbyt częste korzystanie ze składowych (i metod) statycznych jest oznaką niedojrzałości programisty używającego obiektowego języka programowania. Projektant powinien myśleć w kategoriach stanu *obiektu*, a nie stanu *klasy*.

Metody statyczne są najgorsze — ich użycie oznacza bowiem, że programista rozumie proceduralnie, a nie myśli o obiektach, które wykonują różne operacje, bazując na swoim unikalnym stanie.

Ależ oczywiście... Wmawiaj sobie, co tylko chcesz...

Składowa statyczna

O czym?

Co to znaczy antyobiektowe?

Ja NIE jestem zmienną globalną. Nie ma czegoś takiego. Ja istnieję w klasie! To jest całkiem obiektowe, wiesz? KLASA. Nie istnieję w jakimś nieokreślonym miejscu, jestem naturalną częścią stanu obiektu; jedyna różnica polega na tym, że jestem współużytkowana przez wszystkie obiekty klasy. A to bardzo efektywne rozwiązanie.

Zaraz, zaraz, chwileczkę! To absolutnie nie jest prawda. Niektóre składowe statyczne mają kluczowe znaczenie dla systemu. A nawet te, które nie są kluczowe, są przynajmniej wygodne.

Czemu tak twierdzisz? I co jest złego w metodach statycznych?

Pewnie, zdaję sobie sprawę, że obiekty powinny koncentrować uwagę na projektowaniu oprogramowania zorientowanego obiektowo; jednak sam fakt, że istnieją bezmyśli programiści, nie oznacza, że... trzeba „wylewać dziecko razem z kodem bajtowym”. Istnieje czas i są sytuacje, w których należy używać składowych statycznych, a kiedy okaże się, że taka składowa jest konieczna, to żadne inne rozwiązanie nie będzie od niej lepsze.



BĄDŹ kompilatorem

Plik przedstawiony na tej stronie stanowi kompletny kod programu. Twoim zadaniem jest stać się kompilatorem i określić, czy przedstawiony program skompiluje się czy nie. Jeśli nie można go skompilować, to jak go poprawić? Jeśli natomiast można go skompilować, to jakie wygeneruje wyniki?



```
class StaticSuper {  
  
    static {  
        System.out.println("Blok bazowego inicjalizatora statycznego");  
    }  
  
    StaticSuper {  
        System.out.println("Konstruktor bazowy");  
    }  
}  
  
public class StaticTester extends StaticSuper {  
    static int rnd;  
  
    static {  
        rnd = (int) (Math.random() * 6);  
        System.out.println("Blok inicjalizatora statycznego " + rnd);  
    }  
  
    StaticTester() {  
        System.out.println("Konstruktor");  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Metoda main");  
        StaticTester st = new StaticTester();  
    }  
}
```

Jeśli program można skompilować, to które z poniższych wyników zostaną wygenerowane w przypadku jego wykonania?

Możliwe wyniki

```
Wiersz polecenia  
T:\>java StaticTester  
Blok bazowego inicjalizatora statycznego 4  
Metoda main  
Blok inicjalizatora statycznego  
Konstruktor bazowy  
Konstruktor
```

Możliwe wyniki

```
Wiersz polecenia  
T:\>java StaticTester  
Blok bazowego inicjalizatora statycznego  
Blok inicjalizatora statycznego 3  
Metoda main  
Konstruktor bazowy  
Konstruktor
```



W tym rozdziale poznaliśmy wspaniały statyczny, świat Javy.
Twoim zadaniem jest określenie, czy każde z poniższych
stwierdzeń jest prawdziwe czy też nie.

👉 PRAWDA CZY FAŁSZ 🙌

1. Aby skorzystać z metod i składowych klasy `Math`, w pierwszej kolejności należy utworzyć obiekt tej klasy.
2. Tworząc konstruktor, można go poprzedzić słowem kluczowym **static**.
3. Metody statyczne nie mają dostępu do składowych reprezentujących stan danego obiektu (określonego przez słowo kluczowe „`this`”).
4. Wywoływanie metod statycznych przy użyciu odwołania do obiektu jest uważane za dobre i właściwe rozwiązanie.
5. Składowe statyczne można wykorzystać do zliczania ilości obiektów danej klasy.
6. Konstruktory są wykonywane przed inicjalizacją składowych statycznych.
7. `ROZMIAR_MAX` byłoby dobrą nazwą składowej statycznej.
8. Blok inicjalizatora statycznego jest wykonywany przed konstruktorem.
9. Jeśli klasa jest oznaczona jako finalna, to także wszystkie jej metody muszą być oznaczone jako finalne.
10. Finalną metodę można przesłonić wyłącznie w przypadku tworzenia klasy potomnej.
11. Nie istnieje klasa reprezentująca wartości logiczne.
12. Klassy reprezentujące typy podstawowe są używane, gdy wartość któregoś z takich typów chcemy potraktować jako obiekt.
13. Metoda `parseXxx` zawsze zwraca łańcuch znaków (obiekt `String`).
14. Klassy formatujące (które w Javie zostały oddzielone od biblioteki wejścia-wyjścia) wchodzą w skład pakietu `java.format`.



Ćwiczenie



Księżycowe magnesiki z kodem

To ćwiczenie może Ci się przydać! Oprócz tego, czego dowiedziałeś się o operowaniu na datach podczas lektury kilku ostatnich stron, będziesz jednak potrzebował kilku dodatkowych informacji... Przede wszystkim — pełnia występują co 29,52 doby, albo coś koło tego. Po drugie, 7 stycznia 2004 roku była pełnia. Twoim zadaniem jest poukładanie poniższych fragmentów kodu i odtworzenie działającego programu, który wygeneruje wyniki przedstawione u dołu strony (wraz z listą kolejnych dat kiedy nastąpi pełnia). (Może się zdarzyć, że nie będziesz potrzebował wszystkich magnesików, oprócz tego, w razie potrzeby, możesz dodać kilka nawiasów klamrowych). A... jeszcze jedno, jeśli mieszkasz w innej strefie czasowej niż nasza, to uzyskasz inne wyniki.

```
long day1 = c.getTimeInMillis();  
c.set(2004,1,7,15,40);
```

```
import static java.lang.System.out;
```

```
static int DAY_IM = 60 * 60 * 24;
```

```
("Pelnia wypada %tc", c));
```

```
Calendar c = new Calendar();
```

```
(c.format
```

```
class PelniaKsiezyca {
```

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

```
day1 += (DAY_IM * 29.52);
```

```
for (int x = 0; x < 60; x++) {
```

```
static int DAY_IM = 1000 * 60 * 60 * 24;
```

```
println
```

```
import java.io.*;
```

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

```
static import java.lang.System.out;
```

```
c.set(2004,0,7,15,40);
```

```
out.println
```

```
c.setTimeInMillis(day1);
```

```
(String.format
```

```
Calendar c = Calendar.getInstance();
```

Wiersz polecenia

```
T:\>java PelniaKsiezyca  
Pelnia wypada Pt lut 06 04:09:27 CET 2004  
Pelnia wypada So mar 06 16:38:15 CET 2004  
Pelnia wypada Pn kwi 05 06:02:03 CEST 2004  
Pelnia wypada Wt maj 04 18:35:51 CEST 2004  
Pelnia wypada Cz cze 03 07:04:39 CEST 2004  
Pelnia wypada Pt lip 02 19:33:27 CEST 2004
```

Rozwiążanie ćwiczenia

BĄDŹ kompilatorem

```
StaticSuper() {
    System.out.println("Konstruktor bazowy");
}
```

StaticSuper to konstruktor i dlatego w swojej sygnaturze musi mieć parę nawiasów – (). Zwróć uwagę, iż zgodnie z tym, co widać na poniższym rysunku, bloki inicjalizatorów statycznych są wykonywane, zanim zostanie wykonany konstruktor kolejkowiek z klas.

Możliwe wyniki

```
ca. Wiersz polecenia

T:\>java StaticTester
Blok bazowego inicjalizatora statycznego
Blok inicjalizatora statycznego 3
Metoda main
Konstruktor bazowy
Konstruktor
```

PRAWDA CZY FAŁSZ

- Aby skorzystać z metod i składowych klasy Math, w pierwszej kolejności należy utworzyć obiekt tej klasy. **Fałsz**
- Tworząc konstruktor, można go poprzedzić słowem kluczowym static. **Fałsz**
- Metody statyczne nie mają dostępu do składowych reprezentujących stan danego obiektu (określonego przez słowo kluczowe „this”). **Prawda**
- Wywoływanie metod statycznych przy użyciu odwołania do obiektu jest uważane za dobre i właściwe rozwiązanie. **Fałsz**
- Składowe statyczne można wykorzystać do zliczania ilości obiektów danej klasy. **Prawda**
- Konstruktory są wykonywane przed inicjalizacją składowych statycznych. **Fałsz**
- ROZMIAR_MAX byłoby dobrą nazwą składowej statycznej. **Prawda**
- Blok inicjalizatora statycznego jest wykonywany przed konstruktorem. **Prawda**
- Jeśli klasa jest oznaczona jako finalna, to także wszystkie jej metody muszą być oznaczone jako finalne. **Fałsz**
- Finalną metodę można przesłonić wyłącznie w przypadku tworzenia klasy potomnej. **Fałsz**
- Nie istnieje klasa reprezentująca wartości logiczne. **Fałsz**
- Klasy reprezentujące typy podstawowe są używane, gdy wartość któregoś z takich typów chcemy potraktować jako obiekt. **Prawda**
- Metoda parseXxx zawsze zwraca łańcuch znaków (obiekt String). **Fałsz**
- Klasy formatujące (które w Javie zostały oddzielone od biblioteki wejścia-wyjścia) wchodzą w skład pakietu java.format. **Fałsz**

Rozwiązywanie księżyckowych magnesików



Rozwiązywanie ćwiczeń

```
import java.util.*;
import static java.lang.System.out;

class PełniaKsiezyc {
    static int DAY_IM = 1000 * 60 * 60 * 24;

    public static void main(String [] args) {
        Calendar c = Calendar.getInstance();
        c.set(2004,0,7,15,40);

        long day1 = c.getTimeInMillis();

        for (int x = 0; x < 60; x++) {
            day1 += (DAY_IM * 29.52);
            c.setTimeInMillis(day1);
            out.println(String.format("Pełnia wypada %tc", c));
        }
    }
}
```

Uwagi do księżyckowych magnesików:

Być może zauważysz, że kilka dat wyznaczonych przez ten program nieco odbiegało od prawidłowych wyników. Te sprawy związane z obliczeniami astronomicznymi są złożone i gdybyśmy chcieli wszystko wyliczać prawidłowo, program byłby zbyt złożony jak na takie ćwiczenie.

Podpowiedź: jednym z problemów który możesz spróbować rozwiązać, są różnice związane ze strefami czasowymi. Czy jesteś w stanie go znaleźć i poprawić?

The screenshot shows a terminal window titled "Wiersz polecenia". The command `T:\>java PełniaKsiezyc` is entered, followed by six lines of output showing the date and time of the next six full moon occurrences in 2004:

```
T:\>java PełniaKsiezyc
Pełnia wypada Pt lut 06 04:09:27 CET 2004
Pełnia wypada So mar 06 16:38:15 CET 2004
Pełnia wypada Pn kwi 05 06:07:03 CEST 2004
Pełnia wypada Wt maj 04 18:35:51 CEST 2004
Pełnia wypada Cz cze 03 07:04:39 CEST 2004
Pełnia wypada Pt lip 02 19:33:27 CEST 2004
```

11. Obsługa wyjątków

Ryzykowne działania

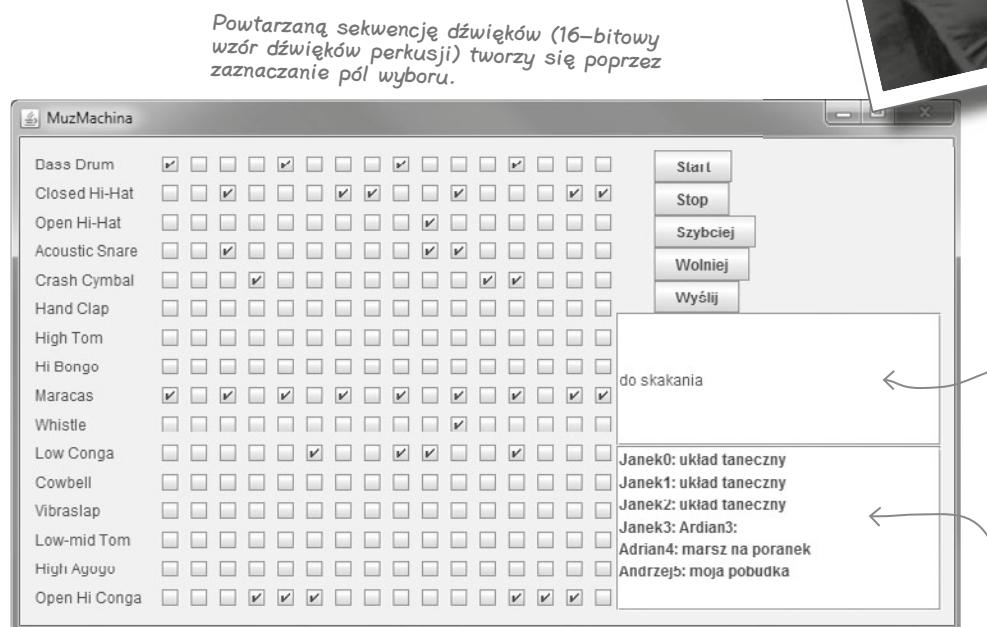


Różne rzeczy się zdarzają. Pliku nie ma tam, gdzie powinien być. Serwer został wyłączony. Niezależnie od tego, jak dobrym jesteś programistą, nie jesteś w stanie kontrolować wszystkiego. Czasami zdarzają się problemy. I to poważne problemy. Pisząc „ryzykowną” metodę, będziesz potrzebował kodu, który „poradzi” sobie w sytuacji, gdy zdarzy się coś złego. Jednak skąd wiadomo, że metoda jest „ryzykowna”? I gdzie należy umieścić kod przeznaczony do obsługi tej *wyjątkowej* sytuacji? Jak na razie w niniejszej książce nie wykonywaliśmy żadnych *naprawdę* ryzykownych operacji. Zdarzały się oczywiście przypadki błędów występujących podczas wykonywania programu, jednak problemy te wynikały zazwyczaj z naszych własnych błędów. Błędów. A te powinniśmy poprawiać w czasie tworzenia programu. Nie, kod służący do obsługi sytuacji wyjątkowych, o jakim będziemy pisać w tym rozdziale, to kod, który *wcale nie musi* zostać wykonany. Kod, który oczekuje, że plik będzie tam, gdzie się spodziewamy, że serwer będzie dostępny, a wątek będzie spokojnie czekać na wznowienie. Jednak musimy poznać zasady jego używania już teraz. Ponieważ w tym rozdziale napiszemy program wykorzystujący „ryzykowny” JavaSound API. Napiszemy odtwarzacz muzyki MIDI.

Stwórzmy program MuzMachina

W tym oraz trzech kolejnych rozdziałach stworzymy kilka aplikacji muzycznych, w tym perkusję cyfrową. W zasadzie przed końcem niniejszej książki stworzymy wersję tego programu obsługującą wielu użytkowników, dzięki czemu będziesz mógł wysyłać swoje solówki do innych „graczy”, podobnie jak w standardowym programie do obsługi internetowych pogawędek. Możesz własnoręcznie napisać cały program, choć możesz także skopiować jego fragmenty związane z graficznym interfejsem użytkownika z „Kodu gotowego do użycia”. No dobrze, nie każdy wydział informatyki będzie potrzebował nowego serwera MuzMachina, jednak robimy to, aby *dowiedzieć się* czegoś więcej o Javie. A pisanie MuzMachiny jest *połączeniem* nauki Javy z niezłą zabawą.

Ostateczna wersja aplikacji MuzMachina wygląda mniej więcej tak:



Twoja wiadomość, która po kliknięciu przycisku „Wyślij”, jest wraz z informacjami o bieżącej kompozycji wysyłana do innych użytkowników.

Wiadomości nadysytane od innych użytkowników. Kliknij jedną z nich, aby wczytać przesłane informacje o kompozycji, a następnie, aby ją odtworzyć, kliknij przycisk Start.

Zaznacz pola wyboru dla każdego z 16 „taktów”. Na przykład, w takcie pierwszym (z 16) zostaną odtworzone dźwięki dla instrumentów Bass drum oraz Maracas, w takcie drugim nie zostaną odtworzone żadne dźwięki, a w takcie trzecim zostaną odtworzone dźwięki dla instrumentów Maracas, Acoustic Snare oraz Closed Hi-Hat. Już pewnie rozumiesz, o co chodzi... Po kliknięciu przycisku *Start* skomponowany utwór będzie odtwarzany w pętli, aż do momentu kliknięcia przycisku *Stop*. W każdej chwili możesz „przechwycić” swoją kompozycję, przesyłając ją na serwer MuzMachina (co oznacza, że także inne osoby będą mogły ją odtworzyć). Możesz także odtworzyć dowolną z odebranych kompozycji, klikając wiadomość przesyłaną wraz z nią.



Zacznijmy do podstaw

Oczywiście zanim zakończymy prace nad aplikacją, będziemy musieli poznać kilka zagadnień, takich jak: sposoby tworzenia interfejsu graficznego przy użyciu biblioteki Swing, sposoby nawiązywania połączenia sieciowego z innym komputerem czy też wykonywanie operacji wejścia-wyjścia, które pozwolą nam coś wysłać na inny komputer.

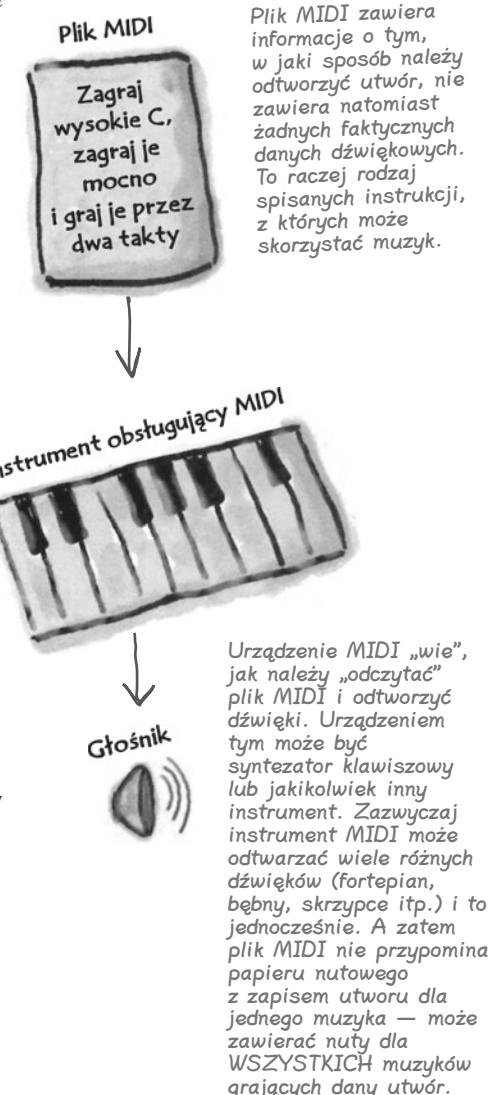
No i oczywiście pozostało interfejs programistyczny JavaSound API. To właśnie od niego zacznijmy niniejszy rozdział. Jak na razie możemy zapomnieć o interfejsie graficznym, o połączeniach sieciowych oraz operacjach wejścia-wyjścia, skoncentrujemy się bowiem na tym, aby odtworzyć muzykę MIDI. I nie przejmuj się, jeśli nie wiesz nic o technologii MIDI albo o odczytywaniu i odtwarzaniu muzyki. Wszystko, czego musisz się dowiedzieć, znajdziesz w tym rozdziale.

JavaSound API

JavaSound to grupa klas oraz interfejsów, które zostały dodane do Javy, zaczynając od wersji 1.3 języka. Nie stanowią one niezależnego dodatku, lecz wchodzą w skład biblioteki Javy dostarczanej w standardowej wersji języka — J2SE. Biblioteka JavaSound została podzielona na dwie części: MIDI oraz Sampled. W niniejszej książce będziemy wykorzystywać tylko pierwszą z nich. MIDI to skrót angielskich słów Musical Instrument Digital Interface (intrefejs cyfrowy dla urządzeń muzycznych); jest to standardowy protokół umożliwiający współdziałanie różnego typu elektronicznych urządzeń muzycznych, jednak w przypadku naszej aplikacji możesz wyobrażać sobie MIDI jako coś w rodzaju papieru nutowego, który można włożyć do jakiegoś urządzenia; to z kolei możesz sobie wyobrażać jako bardzo zaawansowane technologicznie pianino. Innymi słowy, dane MIDI nie zawierają żadnych dźwięków, zawierają natomiast instrukcje dla instrumentu MIDI, które pozwolą mu na wygenerowanie muzyki. Można podać także inną analogię — wyobraź sobie, że plik MIDI przypomina dokument HTML, a instrument muzyczny przetwarzający pliki MIDI (czyli, na przykład, odtwarzający je) przypomina przeglądarkę WWW.

Dane MIDI określają, co należy zrobić (zagraj wysokie C, a oto, jak mocno należy zagrać ten dźwięk i jak długo ma on trwać itd.), jednak nie zawierają żadnych informacji o faktycznym dźwięku, który usłyszysz. Dane MIDI nie „wiedzą”, jak ma brzmieć flet, fortepian ani gitara Jimmiego Hendrixa. Aby uzyskać dźwięk, będziemy potrzebować instrumentu (urządzenia MIDI) potrafiącego odczytywać i odtwarzać pliki MIDI, jednak takie urządzenie zazwyczaj bardziej przypomina cały zespół instrumentów lub orkiestrę, a instrument może być urządzeniem fizycznym, takim jak elektroniczny syntezator używany przez muzyków rockowych lub też urządzeniem całkowicie programowym, istniejącym w Twoim komputerze.

W naszej aplikacji będziemy używać wyłącznie instrumentu programowego, dostarczanego wraz z Javą. Jest on nazywany syntezatorem (nекоторые называют его также syntezatorem programowym), gdyż tworzy dźwięki. Dźwięki, które możesz usłyszeć.



Ale to wyglądało na takie łatwe

Przede wszystkim potrzebujemy sekwensera

Zanim będziemy mogli zagrać jakikolwiek dźwięk, potrzebny nam będzie sekwenser — czyli obiekt Sequencer. Obiekt ten odczytuje wszystkie dane zapisane w pliku MIDI i przesyła je do odpowiedniego urządzenia. To właśnie on umożliwia *zagranie* muzyki. Sekwenser może wykonywać wiele różnych operacji, jednak w tej książce będziemy go używali wyłącznie jako urządzenia do odtwarzania. Jak gdyby był on odtwarzaczem CD wchodząącym w skład Twojej „wieży” i wyposażonym w kilka dodatkowych możliwości. Klasa Sequencer wchodzi w skład pakietu javax.sound.midi (należy on do standardowej biblioteki Javy od wersji 1.3). A zatem na samym początku upewnijmy się, że możemy stworzyć (lub pobrać) obiekt Sequencer.

```
import javax.sound.midi.*;  Importujemy pakiet javax.sound.midi.
```

```
public class MuzaTester1 {  
    public void graj() {  
        Sequencer sekwenser = MidiSystem.getSequencer();  
        System.out.println("Mamy sekwenser!");   
    } // koniec graj
```

Potrzebujemy sekwensera czyli obiektu Sequencer. To podstawowy element używanego przez nas urządzenia (instrumentu) MIDI. Sekwenser to jest „to coś”, co zamienia dane MIDI na „utwór muzyczny”. Nie będziemy jednak tworzyć nowego sekwensera — musimy poprosić klasę MidiSystem o zwrócenie istniejącego.

```
    public static void main(String[] args) {  
        MuzaTester1 mt = new MuzaTester1();  
  
        mt.graj();  
    } // koniec main  
} // koniec klasy
```

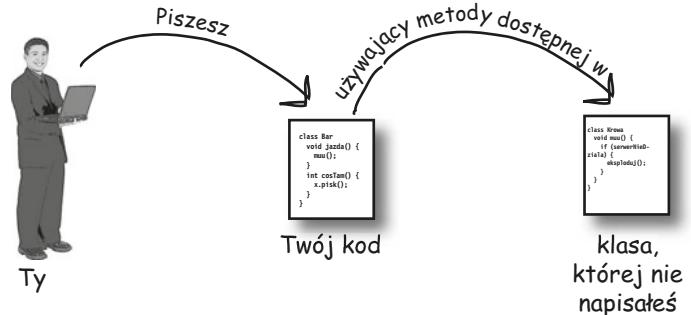
Coś jest nie tak!

Kod nie chce się skompilować! Kompilator twierdzi, że wystąpił jakiś „niezagłoszony wyjątek”, który należy przechwycić lub zadeklarować.

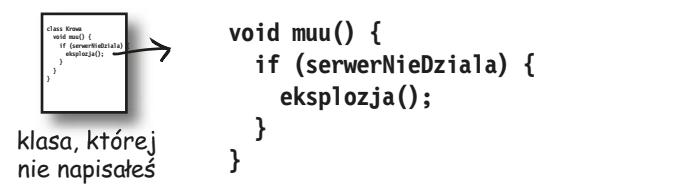


Co się dzieje, gdy metoda, którą chcesz wywołać (prawdopodobnie należąca do klasy, której nie napisałesz), jest „ryzykowana”?

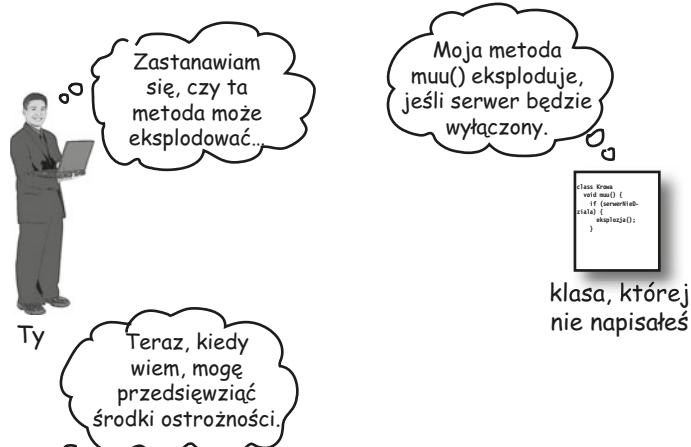
- Założmy, że chcesz wywołać metodę dostępną w klasie, której nie napisiałeś.



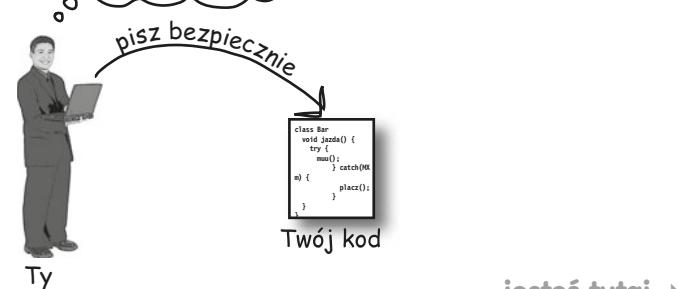
- Ta metoda robi coś ryzykownego, coś, co podczas wykonywania programu może nie zadziałać.



- Musisz wiedzieć, że wywoływana metoda jest niebezpieczna.



- Następnie tworzysz kod, który obsługuje niepowodzenie, jeśli faktycznie się zdarzy. Musisz być przygotowany, tak na wszelki wypadek.



Kiedy sprawy mogą pójść nie tak

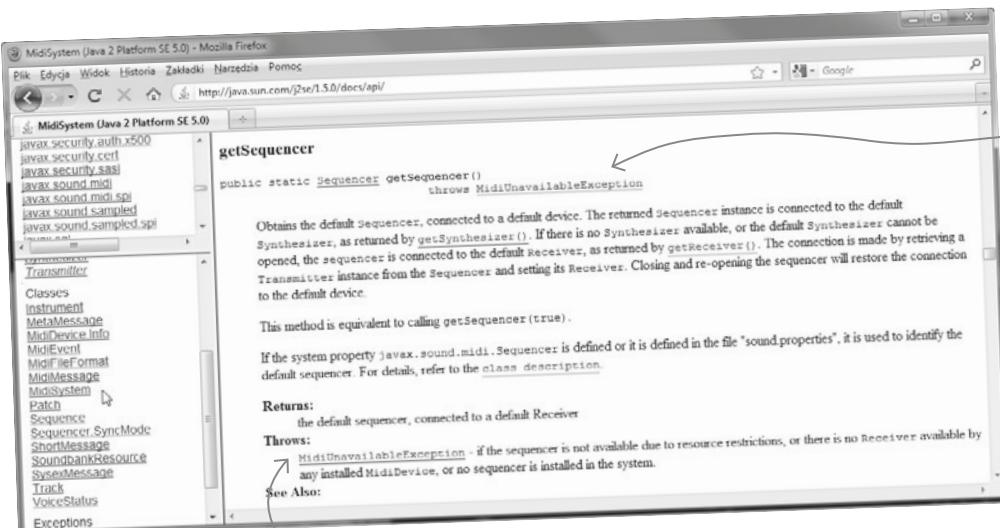
W Javie metody używają wyjątków, aby przekazać informację:

„Stało się coś złego. Zawiodłam”.

Mechanizm obsługi wyjątków Javy jest eleganckim i dobrze udokumentowanym sposobem obsługi „sytuacji wyjątkowych” zachodzących podczas działania programu. Dodatkowo pozwala on na umieszczanie całego kodu służącego do obsługi wyjątków w jednym, łatwym do znalezienia miejscu. Mechanizm ten bazuje na *wiedzy*, że wywoływana metoda jest ryzykowana (to znaczy, że *może* zgłosić wyjątek), dzięki czemu jesteś w stanie napisać kod, który obsłuży taką ewentualność. *Wiedząc*, że wywołanie konkretnej metody może spowodować zgłoszenie wyjątku, możesz być *przygotowanym* na wystąpienie problemu będącego przyczyną wyjątku, a nawet możesz go *rozwiązać*.

A zatem, jak można się dowiedzieć, że metoda zgłasza wyjątek? Informuje o tym klauzula throws umieszczona w deklaracji metody.

Metoda getSequencer() podejmuje pewne ryzyko. Istnieje prawdopodobieństwo, że podczas działania programu wykonywane przez nią operacje zakończą się niepomyślnie. A zatem metoda ta musi „zadeklarować” ryzyko, jakie bierzesz na siebie, wywołując ją.



Dokumentacja API informuje, że metoda `getSequencer()` może zgłosić wyjątek, a konkretnie wyjątek `MidiUnavailableException`. Metoda musi deklarować wyjątki, które może zgłaszać.

Ten fragment dokumentacji informuje, KIEDY wyjątek może zostać zgłoszony. W tym przypadku jego przyczyną mogą być ograniczenia w dostępie do zasobów (co może po prostu oznaczać, że sekwenser jest aktualnie używany).

Kompilator musi wiedzieć, że TY wiesz, że wywołujesz ryzykowną metodę

Kompilator wyraźnie się „zrelaksuje”, jeśli umieścisz wywołanie ryzykownej metody w czymś, co jest określane jako blok **try–catch**.

Blok try–catch informuje kompilator, że wiesz o sytuacji wyjątkowej, która może się zdarzyć w wywoływanej metodzie, oraz że jesteś przygotowany, aby tę sytuację obsłużyć. Kompilatora nie interesuje, w jaki sposób obsłużysz tę sytuację, zwraca on uwagę wyłącznie na sam fakt, że ją obsłużysz.

```
import javax.sound.midi.*;

public class MuzaTester1 {
    public void graj() {

        try {
            Sequencer sekwenser = MidiSystem.getSequencer(); ← Umieść ryzykowne operacje
            System.out.println("Mamy sekwenser!");
        } catch (MidiUnavailableException ex) { ← w bloku „try”.
            System.out.println("Masz problem");
        }
    } // koniec graj

    public static void main(String[] args) {
        MuzaTester1 mt = new MuzaTester1();
        mt.graj();
    } // koniec main
} // koniec klasy
```

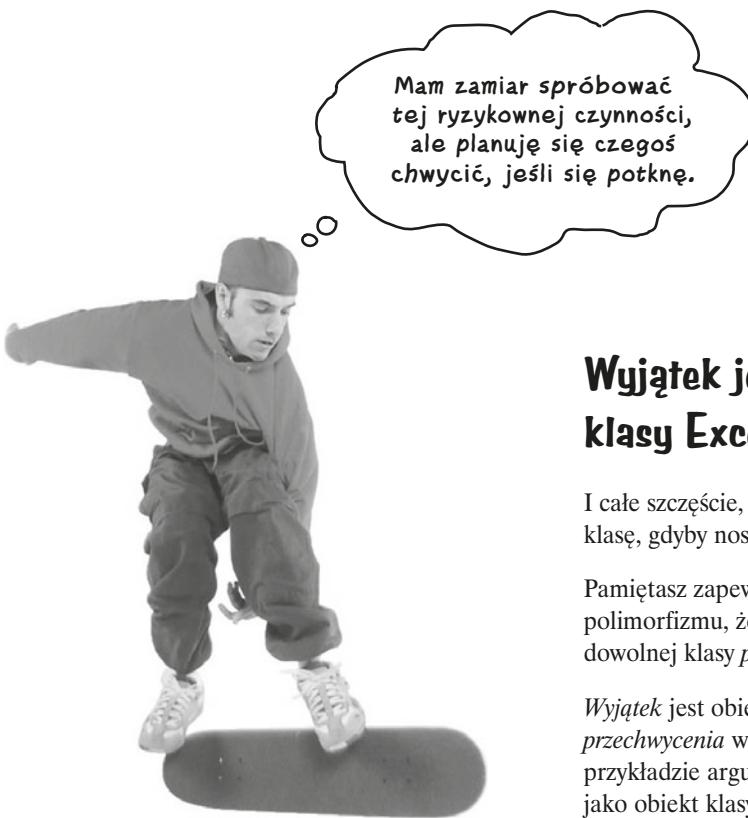
Drogi Kompilatorze,

Wiem, że ryzykuję jednak czy nie uważasz, że warto podjąć to ryzyko?
Cóż mogę zrobić?

Podpisano. Dziki z Waikiki

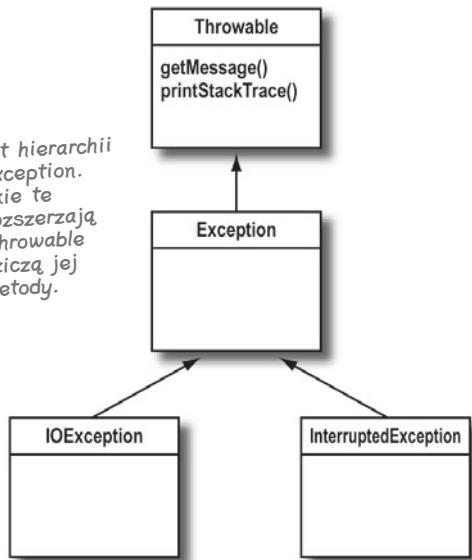
Drogi Dziki,

Zycie jest krótkie (zwłaszcza na stercie).
Rzykuj. Spróbuj swych sił. Jeśli jednak okazałoby się, że nie podołasz, to nie zapomnij przechwycić wszystkich problemów zanim spowodują tragiczne konsekwencje.



Nie próbuj robić tego w domu.

Fragment hierarchii klas Exception. Wszystkie te klasy rozszerzają klasę Throwable i dziedziczą jej dwie metody.



Wyjątek jest obiektem... klasy Exception

I całe szczęście, bo znacznie trudniej byłoby zapamiętać tę klasę, gdyby nosiła inną nazwę, na przykład Kalafior.

Pamiętasz zapewne z rozdziałów poświęconych zagadnieniom polimorfizmu, że obiektem typu `Exception` może być obiekt dowolnej klasy *potomnej* klasy `Exception`.

Wyjątek jest obiektem, a zatem to, co uzyskujemy w wyniku przechwycenia wyjątku, jest obiektem. W poniższym przykładzie argument bloku `catch` został zadeklarowany jako obiekt klasy `Exception`, a parametr, w którym wyjątek zostanie zapisany, nosi nazwę `ex`.

```
try {
    // zrób coś ryzykownego
} catch (Exception ex) {
    // spróbuj rozwiązać problem
}
```

To przypomina deklarowanie argumentu metody.

Ten kod zostanie wykonany wyłącznie w przypadku, gdy zostanie zgłoszony wyjątek `Exception`.

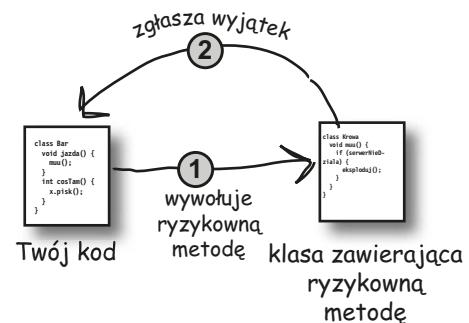
Kod, jaki należy umieścić wewnętrz boku `catch`, zależy od zgłoszonego wyjątku. Na przykład, jeśli serwer będzie niedostępny, to wewnętrz bloku `catch` możesz spróbować nawiązać połączenie z innym serwerem. Jeśli natomiast nie będzie potrzebnego pliku, możesz poprosić użytkownika, aby pomógł Ci go odszukać.

Jeśli to Twój kod przechwytuje wyjątek, to czuj kod go zgłasza?

Pisząc programy w Javie, znacznie więcej czasu spędzasz na *obsługiwaniu wyjątków*, niż na ich *tworzeniu i zgłoszaniu*. Jak na razie wystarczy, abyś wiedział, że jeśli kod *wywołuje ryzykowną metodę* — czyli metodę, która deklaruje wyjątek — to właśnie ta metoda zgłasza wyjątek do *Twojego kodu*.

W rzeczywistości Ty sam możesz być autorem obu tych klas. Autorstwo kodu nie ma w tym przypadku większego znaczenia. Liczy się tylko to, aby wiedzieć, która metoda zgłasza wyjątek i która go *przechwytuje*.

Jeśli ktoś tworzy kod, który może zgłaszać wyjątki, musi te wyjątki *zadeklarować*.



① Ryzykowna metoda zgłaszająca wyjątek:

```

public void zaryzykuj() throws ZłyWyjatek {
    if (porzucWSzelkieNadzieje) {
        throw new ZłyWyjatek();
    }
}
  
```

Tworzymy obiekt i zgłaszymy wyjątek.

Ta metoda **MUSI ostrzec świat** (poprzez odpowiednią deklarację), że zgłasza wyjątek klasy *ZłyWyjatek*.

Jedna metoda **przechwytuje** to, co inną zgłosi. Wyjątek jest zawsze przekazywany z powrotem do metody wywołującej.

Metoda zgłaszająca wyjątek musi zadeklarować, że może go zgłosić.

② Twój kod, który wywołuje ryzykowną metodę:

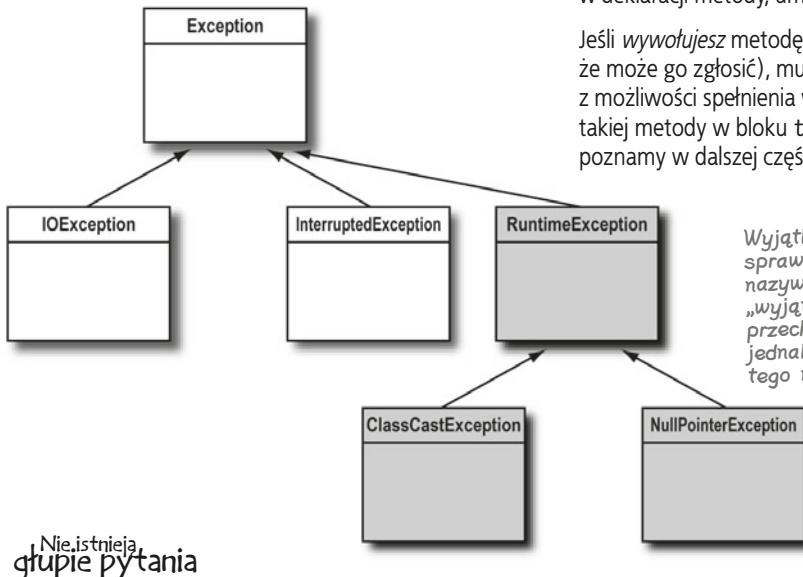
```

public void OdpukajWNiemalowaneDrewno() {
    try {
        jakiśObiekt.zaryzykuj();
    } catch (ZłyWyjatek ex) {
        System.out.println("Ratunkuuu!!!");
        ex.printStackTrace();
    }
}
  
```

Jeśli nie jesteś w stanie rozwiązać problemu, który doprowadził do zgłoszenia wyjątku, to **PRZYNAJMNIĘJ** wyświetl aktualny stan stosu, wykorzystując w tym celu metodę `printStackTrace()`, którą dziedziczą wszystkie wyjątki.

Wyjątki sprawdzane i niesprawdzane

Wyjątki, które NIE są klasami potomnymi klasy RuntimeException, są sprawdzane przez kompilator. Dlatego też nazywamy je „wyjątkami sprawdzanymi”.



P: Zaraz, chwileczkę! Jak to się stało, że to jest pierwszy przypadek, w którym musimy użyć bloku try—catch? A co z wyjątkami, które pojawiały się już w przykładach przedstawionych we wcześniejszych rozdziałach, na przykład z wyjątkiem `NullPointerException` albo z wyjątkiem związanym z dzieleniem przez zero. Zdarzyło mi się nawet, że pojawił się wyjątek `NumberFormatException` zgłoszony przez metodę `Integer.parseInt()`. Dlaczego tych wyjątków nie musimy przechwytywać?

O: Kompilator zwraca uwagę na wszystkie klasy potomne klasy `Exception` za wyjątkiem wyjątków specjalnego typu — `RuntimeException` — tak zwanych „wyjątków czasu wykonywania programu”. Każda klasa rozszerzająca `RuntimeException` ma „wolną drogę”. Wyjątki tego typu mogą być zgłaszane w dowolnym miejscu kodu i nie trzeba ich deklarować ani przechwytywać przy użyciu bloków try—catch. Kompilator nie zaprzata sobie głowy sprawdzaniem, czy metoda deklaruje, że może zgłosić wyjątek `RuntimeException`, ani aby kod wywołujący taką metodę potwierdza, że może taki wyjątek przechwycić i obsłużyć.

Kompilator sprawdza wszystkie wyjątki za wyjątkiem wyjątków klasy `RuntimeException`.

Kompilator gwarantuje, że:

Jeśli zgłaszasz wyjątek w swoim kodzie, musisz ten fakt oznajmić w deklaracji metody, umieszczając w niej słowo kluczowe `throws`.

Jeśli wywołujesz metodę zgłaszącą wyjątek (czyli metodę, która deklaruje, że może go zgłosić), musisz potwierdzić, że wiesz o tej możliwości. Jedną z możliwości spełnienia wymagań kompilatora jest umieszczenie wywołania takiej metody w bloku `try-catch`. (Jest jeszcze inny sposób, który poznamy w dalszej części tego rozdziału).

Wyjątki klasy `RuntimeException` NIE są sprawdzane przez kompilator. Są one nazywane (co jest bardzo zaskakujące) „wyjątkami niesprawdzanymi”. Można zgłaszać, przechwytywać i deklarować wyjątki tego typu, jednak nie trzeba tego robić — kompilator i tak tego nie sprawdza.

P: Będę dociekliwy. DLACZEGO kompilator nie zwraca uwagi na te wyjątki? Czy nie mogą one doprowadzić do przerwania działania programu?

O: Większość wyjątków `RuntimeException` jest spowodowanych problemami związanymi z logiką działania Twojego kodu, a nie z warunkami, które nie są spełnione podczas wykonywania programu i to w taki sposób, którego nie jesteś w stanie przewidzieć ani mu zapobiec. Nie możesz zagwarantować, że niezbędny plik będzie na miejscu. Nie możesz zagwarantować, że serwer będzie dostępny w sieci. Jednak możesz zagwarantować, że program nie będzie pobierać danych z komórek, których indeksy wykraczają poza zakres tablicy (właśnie do tego służy atrybut `.length`).

Zatem CHCESZ, aby wyjątki `RuntimeException` były zgłaszane podczas tworzenia i testowania programu, lecz jednocześnie chciałbyś uniknąć konieczności umieszczania kodu w blokach `try-catch` i narażania się na wszystkie związane z tym konsekwencje tylko po to, aby przechwycić wyjątek, który w ogóle nie powinien się pojawić.

Blok try—catch służy do obsługi sytuacji wyjątkowych, a nie błędów i usterek w kodzie. Bloków catch należy używać, aby podjąć próbę wyjścia z sytuacji, które mogą zakończyć się porażką. Lub, w ostatczności, aby wyświetlić stosowny komunikat i bieżącą zawartość stosu, dzięki czemu ktoś będzie w stanie określić, co się stało.



CELNE SPOSTRZEŻENIA

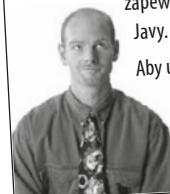
- Metoda może zgłaszać wyjątek, jeśli podczas działania programu zdarzy się coś nieprzewidzianego.
- Wyjątek zawsze jest obiektem klasy `Exception`. (A jak zapewne pamiętasz z rozdziałów poświęconych zagadnieniom polimorfizmu, oznacza to, że jest on obiektem klasy, w której drzewie dziedziczenia znajduje się klasa `Exception`).
- Kompilator NIE zwraca uwagi na wyjątki typu **`RuntimeException`**. Takich wyjątków nie trzeba ani deklarować, ani przechwytywać przy użyciu bloków `try-catch` (choć oczywiście możesz to robić).
- Wszystkie wyjątki, na które kompilator zwraca uwagę, są nazywane „wyjątkami sprawdzanymi”, co w rzeczywistości oznacza, że są to wyjątki *sprawdzane przez kompilator*. Jedynie wyjątki `RuntimeException` nie są sprawdzane. Wszystkie pozostałe wyjątki muszą zostać potwierdzone w kodzie, zgodnie z określonymi regułami.
- Metoda zgłasza wyjątek przy użyciu słowa kluczowego **`throw`**, po którym tworzony jest nowy obiekt wyjątku: **`throw new BrakKawyException();`**
- Metoda, która *może* zgłosić wyjątek, ***musi*** o tym poinformować przy użyciu deklaracji **`throws KlasaWyjątku`**.
- Jeśli Twój kod wywołuje metodę, która może zgłosić wyjątek sprawdzany, musi zapewnić kompilator, że zostały podjęte odpowiednie środki zabezpieczające.
- Jeśli jesteś przygotowany do obsługi wyjątku, umieść wywołanie w bloku `try-catch`, a kod obsługujący wyjątek w bloku `catch`.
- Jeśli nie jesteś przygotowany do obsługi wyjątku, to i tak możesz spełnić wymagania kompilatora, oficjalnie „rezygnując” z jego obsługi. To zagadnienie zostało opisane w dalszej części rozdziału.

Wspomóż metapoznanie

Jeśli próbujesz się nauczyć czegoś nowego, to niech będzie to ostatnia rzecz, jaką starasz się zapamiętać przed zaśnięciem. A zatem, jeśli odłożysz tę książkę (zakładając, że w ogóle będziesz w stanie oderwać się od niej), to nie czytasz potem niczego, co byłoby bardziej wyzywające intelektualnie od opakowania płatków kukurydzianych. Twój mózg potrzebuje czasu na przetworzenie tego, co przeczytałeś i czego się nauczyłeś. A to może zająć mu nawet kilka godzin. Jeśli spróbujesz „wepchnąć” do głowy coś nowego, to pewne informacje o Javie mogą się z niej „ulotnić”.

Oczywiście nie dotyczy to nabycia umiejętności fizycznych. Odbycie wieczornego treningu kickboxingu zapewne nie wpłynie na wcześniejszą naukę Javy.

Aby uzyskać jak najlepsze rezultaty, czytaj tę książkę (albo przynajmniej pooglądaj obrazki) przed udaniem się na spoczynek.

**Zaostrz ołówek**

Jak myślisz, które z wymienionych obok operacji mogłyby spowodować zgłoszenie wyjątku? Oczywiście, chodzi nam tylko o sytuacje, których nie możesz kontrolować z poziomu kodu. Pierwszą z takich operacji zaznaczyliśmy sami.

(Gdyż była najprostsza).

Operacje, które chcemy wykonać

- nawiązanie połączenia ze zdalnym serwerem
- odczyt wartości z komórki poza zakresem tablicy
- wyświetlenie okna na ekranie komputera
- pobranie informacji z bazy danych
- sprawdzenie, czy plik znajduje się tam, gdzie oczekujemy
- utworzenie nowego pliku
- odczytanie znaku z wiersza poleceń

Co może się zdarzyć?

serwer jest niedostępny

Sterowanie przepływem w blokach try-catch

Wywołanie ryzykownej metody może się zakończyć na dwa sposoby — pomyślnie, a w takim przypadku blok `try` zostaje normalnie zakończony, bądź też metoda może zgłosić wyjątek, który zostanie przekazany do metody wywołującej.

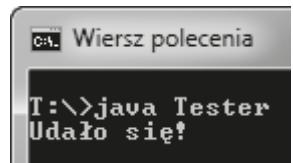
Jeśli wykonanie metody zakończy się pomyślnie

(metoda `zaryzykuj()` nie zgłasza wyjątku)

Najpierw wykonywany jest kod umieszczony w bloku `try`, a następnie kod umieszczony poniżej bloku `catch`.

```
try {  
    ① Test t = x.zaryzykuj;  
    int b = f.getLiczba;  
  
} catch (Exception ex) {  
    System.out.println("Nie udało się");  
}  
  
② → System.out.println("Udało się!");
```

Kod umieszczony w bloku `catch` nigdy nie zostanie wykonany.



Jeśli próba się NIE powiedzie

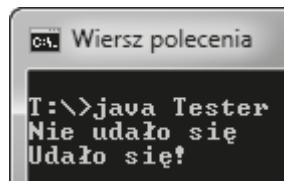
(gdyż metoda `zaryzykuj()` zgłosi wyjątek)

Podczas wykonywania bloku `try` metoda `zaryzykuj()` zgłasza wyjątek, zatem cała reszta bloku zostaje pominięta.

Następnie wykonywany jest blok `catch` oraz dalsza część metody.

```
try {  
    ① Test t = x.zaryzykuj();  
    int b = f.getLiczba();  
  
} catch (Exception ex) {  
    ② → System.out.println("Nie udało się");  
}  
  
③ → System.out.println("Udało się!");
```

Dalsza część bloku `try` nigdy nie zostanie wykonana, co jest ze wszelkich miar właściwe, gdyż zaledwie ona od tego, czy wywołanie metody `zaryzykuj()` zakończy się sukcesem.



Finally — blok kodu, który musi zostać wykonany *niezależnie od wszystkiego*

Jeśli chcesz coś upiec, zaczynasz od włączenia piekarnika.

Jeśli próba pieczenia zakończyła się totalną **porażką**, *musisz wyłączyć piekarnik*.

Jeśli próba pieczenia zakończyła się **sukcesem**, *musisz wyłączyć piekarnik*.

*Musisz wyłączyć piekarnik **niezależnie od wszystkiego**!*

Blok finally służy do umieszczania kodu, który musi zostać wykonany niezależnie od tego, czy wyjątek został zgłoszony czy też nie.

```
try {
    wlaczPiekarnik();
    x.piecz();
} catch (PieczenieException ex) {
    ex.printStackTrace();
} finally {
    wlaczPiekarnik();
}
```

Gdyby nie było bloku **finally**, wywołanie metody `wlaczPiekarnik()` musiałbyś umieścić zarówno w bloku `try`, jak i w bloku `catch`, gdyż piekarnik **trzeba wyłączyć niezależnie od wszystkiego**. Blok `finally` umożliwia umieszczenie w **jednym miejscu** całego ważnego kodu kończącego ryzykowne operacje i uniknięcie konieczności jego powielania.

```
try {
    wlaczPiekarnik();
    x.piecz();
    wlaczPiekarnik();
} catch (PieczenieException ex) {
    ex.printStackTrace();
    wlaczPiekarnik();
}
```



Jeśli wykonanie bloku try zakończy się niepowodzeniem (czyli zostanie zgłoszony wyjątek), to sterowanie jest natychmiast przenoszone do bloku catch. Po zakończeniu realizacji bloku catch wykonywany jest blok finally. Kiedy i on zostanie zakończony, wykonywany jest dalszy kod metody.

Jeśli wykonanie bloku try zakończy się pomyślnie (nie zostanie zgłoszony żaden wyjątek), blok catch jest pomijany, a sterowanie zostaje przeniesione do bloku finally. Po jego zakończeniu wykonywany jest dalszy kod metody.

Blok finally zostanie wykonany nawet w przypadku, gdy w bloku try lub catch została umieszczona instrukcja return. W takiej sytuacji wykonywany jest blok finally, a następnie sterowanie wraca do instrukcji return.

Ćwiczenie ze sterowania wykonywaniem programu



Zaostrz ołówek

Sterowanie wykonywaniem programu

```
public class TestWyjatkow {  
  
    public static void main(String[] args) {  
        String test = "nie";  
        try {  
            System.out.println("Początek bloku try");  
            zaryzykuj(test);  
            System.out.println("Koniec bloku try");  
        } catch (StrasznyWyjatek sw) {  
            System.out.println("Straszny wyjątek");  
        } finally {  
            System.out.println("Blok finally");  
        }  
        System.out.println("Koniec metody main");  
    }  
  
    static void zaryzykuj(String test) throws StrasznyWyjatek {  
        System.out.println("Początek ryzykownej metody");  
        if ("tak".equals(test)) {  
            throw new StrasznyWyjatek();  
        }  
        System.out.println("Koniec ryzykownej metody");  
        return;  
    }  
}
```

Przeanalizuj kod programu podany z lewej strony. Jak myślisz, jakie będą wyniki jego wykonania? Jak sądzisz, jakie byłyby te wyniki, gdyby trzeci wiersz kodu zmienić na: `String test = "tak";`? Przyjmij, że `StrasznyWyjatek` jest klasą potomną klasy `Exception`.

Wyniki generowane, gdy test = "nie"

```
static void zaryzykuj(String test) throws StrasznyWyjatek {
    System.out.println("Początek ryzykownej metody");
    if ("tak".equals(test)) {
        throw new StrasznyWyjatek();
    }
    System.out.println("Koniec ryzykownej metody");
    return;
}
```

Wyniki generowane, gdy test = "tak"

Wyzypadku gdy test = "ak"; Pożądany blok finally — Konieczny blok try — Pożądany blok rzykownej metody — Straszny wyzypadku gdy test = "ak"; Pożądany blok finally — Konieczny blok try — Pożądany rzykownej metody main

Czy wspominaliśmy, że metoda może zgłaszać więcej niż jeden wyjątek?

Metoda, jeśli to niezbędne, może zgłaszać więcej niż jeden wyjątek. Jednak w deklaracji metody należy wymienić **wszystkie** sprawdzane wyjątki, które metoda może zgłaszać (choć w przypadku, gdy dwie lub kilka klas wyjątków mają tę samą klasę bazową, to w deklaracji metody można podać jedynie tę klasę bazową).

Przechwytywanie wielu wyjątków

Kompilator zapewnia, że wszystkie sprawdzane wyjątki zgłasiane przez wywoływaną metodę są obsługiwane. Zgrupuj odpowiednie bloki catch poniżej bloku try, umieszczając je jeden po drugim. Czasami kolejność zapisu poszczególnych bloków catch będzie mieć znaczenie, jednak tym zagadnieniem zajmiemy się w dalszej części rozdziału.

```
public class Pranie {
    public void zrobPranie() throws SpodnieException, BieliznaException {
        // kod, który może zgłaszać oba wyjątki
    }
}
```



Ta metoda deklaruje dwa – sprawdź sam – DWA wyjątki.

```
public class Test {
    public void doDziela() {
        Pranie pranie = new Pranie();
        try {
            pranie.zrobPranie();
        } catch (SpodnieException sex) {
            // kod rozwiązuający problem
        }
    }
}
```



Jeśli metoda zrobPranie() zgłosi wyjątek SpodnieException, zostanie on obsłużony w bloku catch deklarującym SpodnieException.

```
} catch (BieliznaException bex) {
    // kod rozwiązujący problem
}
}
```



Jeśli metoda zrobPranie() zgłosi wyjątek BieliznaException, zostanie on obsłużony w bloku catch deklarującym BieliznaException.

Wyjątki są polimorficzne

Pamiętaj, że wyjątki są obiektami. Nie ma w nich absolutnie niczego szczególnego, oczywiście oprócz faktu, że można je zgłaszać. A zatem można się do nich odwoływać przy wykorzystaniu polimorfizmu, podobnie jak do wszystkich innych obiektów. Na przykład, obiekt `BieliznaException` można zapisać w odwołaniu klasy `OdziezException`. Podobnie obiekt `SpodnieException` można zapisać w odwołaniu klasy `Exception`. Pewnie rozumiesz, o co chodzi. Zaleta tej możliwości polega na tym, że metoda nie musi deklarować wszystkich możliwych wyjątków, które może zgłaszać — może ograniczyć się do zadeklarowania jedynie ich wspólnej klasy bazowej. Dokładnie to samo dotyczy bloków `catch` — nie trzeba tworzyć odrębnych bloków `catch` dla każdego ze zgłaszanych wyjątków, jeśli jeden z bloków (lub kilka z nich) może obsługiwać dowolny ze zgłaszanych wyjątków.

- W DEKLARACJI metody zamiast klas wszystkich zgłaszanych wyjątków można podać ich wspólną klasę bazową.**



```
public void zrobPranie() throws OdziezException {
```



Zadeklarowanie wyjątku typu `OdziezException` pozwala na zgłaszanie wyjątków dowolnych klas potomnych. Oznacza to, że metoda `zrobPranie()` może zgłaszać wyjątki `BieliznaException`, `KoszulkaException` oraz `KoszulaWyjsciowaException` bez konieczności jawnego deklarowania każdego z tych wyjątków.

- Można przechwycić zgłoszony wyjątek, wykorzystując w tym celu jego klasę bazową.**

```
try {
    pranie.zrobPranie();
} catch (OdziezException oex) {
    // kod rozwiązuający problem
}
```



Umożliwia przechwytywanie wyjątku dowolnej klasy potomnej `OdziezException`

```
try {
    pranie.zrobPranie();
} catch (KoszulkaException kex) {
    // kod rozwiązujący problem
}
```



Umożliwia przechwytywanie wyłącznie wyjątków `KoszulkaException` oraz `KoszulaWyjsciowaException`

To, że MOŻESZ przechwytywać wszystkie wyjątki przy użyciu jednego, polimorficznego bloku catch, wcale nie oznacza, że POWINIENIEŚ tak robić.

Mozesz stworzyć swój kod obsługujący wyjątki w taki sposób, aby składał się on tylko z jednego bloku catch. W tym celu w bloku catch należy podać klasę bazową Exception, dzięki czemu, taki blok będzie w stanie przechwycić każdy zgłoszony wyjątek.

```
try {
    pranie.zrobPranie();

} catch (Exception ex) {
    // kod rozwiązuający problem... ←
    Ale JAKI problem? Ten blok catch będzie
    przechwytywać dowolne, czyli WSZYSTKIE,
    wyjątki, zatem nie będziesz automatycznie
    wiedzieć, co poszło nie tak.
}
```

Stwórz osobne bloki catch dla wszystkich wyjątków, które musisz obsługiwać w unikalny sposób.

Na przykład, jeśli program w inny sposób postępuje w przypadku zgłoszenia wyjątku KoszulkaException niż w razie wystąpienia wyjątku BieliznaException, to dla każdego z tych wyjątków stwórz odrębny blok catch. Jeśli jednak wszystkie pozostałe wyjątki OdziezException traktujesz w taki sam sposób, możesz je obsługiwać w jednym miejscu, dodając na samym końcu sekwencji bloków catch blok deklarujący obiekt OdziezException.

```
try {
    pranie.zrobPranie();

} catch (KoszulkaException kex) {
    // rozwiązanie problemu KoszulkaException
    ← Wyjątki KoszulkaException oraz
    BieliznaException muszą zawierać
    unikalny kod, a zatem w ich przypadku
    należy użyć odrębnych bloków catch.

} catch (BieliznaException bex) {
    // rozwiązanie problemu BieliznaException
    ←

} catch (OdziezException oex) {
    // rozwiązanie wszystkich pozostałych problemów
    ← Wszystkie pozostałe wyjątki
    OdziezException są obsługiwane
    w tym bloku.
}
```

Kolejność wielu bloków catch

**W przypadku użycia wielu bloków catch
należy je uporządkować ze względu na zakres
— od najmniejszego do największego**



Ten blok catch przechwytyuje wyjątki `KoszulkaException`, lecz wszystkie pozostałe wyjątki przechodzą dalej.

`catch(KoszulkaException kex)`



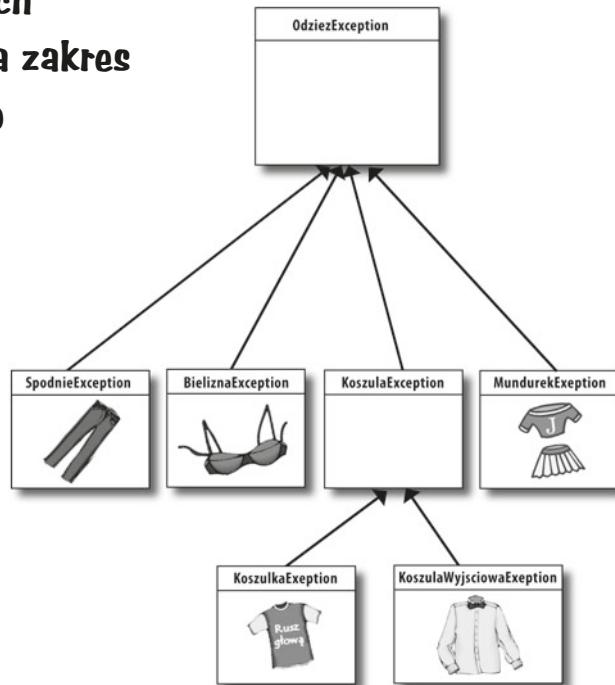
Wyjątki `KoszulkaException` nigdy nie dotrą do tego bloku catch, natomiast zostaną w nim obsłużone wszystkie pozostałe wyjątki `KoszulaException`.

`catch(KoszulaException lex)`



W tym bloku catch obsługiwane są wszystkie wyjątki `OdziezException`, wyjątki `KoszulkaException` oraz `KoszulaException` nigdy do niego nie dotrą.

`catch(OdziezException oex)`



Im wyższe miejsce zajmuje klasa w drzewie dziedziczenia, tym większy jest zakres wyjątków przechwytywanych w bloku catch. Z kolei, przesuwając się ku dołowi drzewa dziedziczenia — ku coraz to bardziej wyspecjalizowanym klasom `Exception` — zakres przechwytywanych wyjątków staje się coraz mniejszy. To nic innego jak zwykły, stary polimorfizm.

Blok catch dla klasy `KoszulaException` może przechwytywać wyjątki `KoszulkaException` oraz `KoszulaWyjsciowaException` (oraz wszystkie przyszłe klasy potomne klasy `KoszulaException`). Jeszcze większy zakres ma blok catch dla klasy `OdziezException` (gdź istnieje więcej obiektów, do których można się odwołać przy użyciu klasy `OdziezException`). Może on przechwytywać wyjątki klasy `OdziezException` oraz wszystkich jej klas potomnych — `SpodnieException`, `MundurekException`, `BieliznaException` oraz `KoszulaException`. „Matką” wszystkich argumentów, jakie można deklarować w blokach catch, jest klasa **Exception** — pozwala ona na przechwytywanie *dowolnych* wyjątków, w tym także wyjątków niesprawdzanych. Z tego powodu prawdopodobnie będziesz jej używać jedynie podczas testowania.

Nie można umieszczać bloków o większym zakresie nad blokami o mniejszym zakresie

No... w zasadzie można, ale takiego kodu nie da się skompilować.

Bloki catch nie przypominają przeciążonych metod, w przypadku których wybierana jest metoda, której sygnatura najlepiej pasuje do wywołania.

W przypadku bloków catch wirtualna maszyna Javy zaczyna od pierwszego z nich i kontynuuje poszukiwania aż do momentu odnalezienia bloku, którego zakres jest wystarczająco szeroki (czyli którego klasa znajduje się wystarczająco wysoko w hierarchii dziedziczenia), aby przechwycić wyjątek. Jeśli pierwszym blokiem będzie **catch(Exception ex)**, kompilator będzie doskonale wiedział, że dodawanie kolejnych bloków nie ma sensu, gdyż żadne wyjątki do nich nie dotrą.

W przypadku stosowania wielu bloków catch wielkość ich zakresu ma znaczenie. Blok o największym zakresie powinien być umieszczony najniżej. W przeciwnym przypadku bloki o mniejszym zakresie staną się bezużyteczne.

Nie rób tak!

```
try {
    pranie.zrobPranie();
}

} catch(OdziezException oex) {
    // rozwiąż problem OdziezException

}

} catch(BieliznaException bex) {
    // rozwiąż problem BieliznaException

}

} catch(KoszulaException kex) {
    // rozwiąż problem KoszulaException
}
```



Bloki różnych klas, lecz o tym samym zakresie, mogą być umieszczane w dowolnej kolejności, gdyż nie będą przechwytywać wyjątków, które powinny trafić do innego bloku.

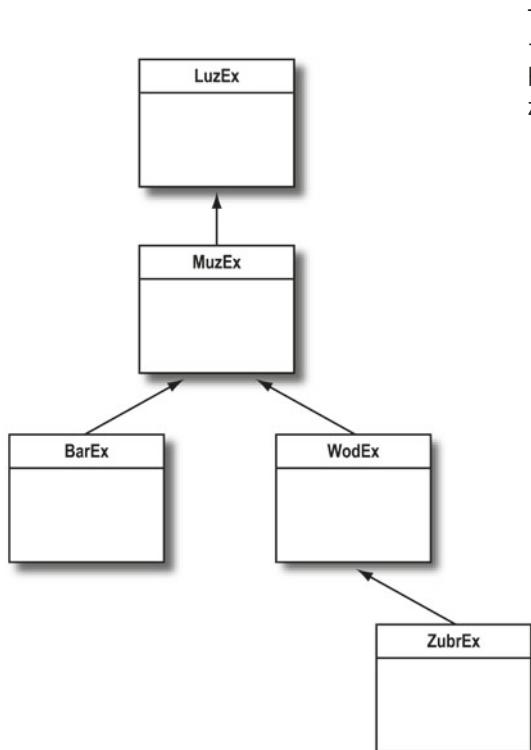
Moglibyś umieścić blok catch dla klasy KoszulaException przed blokiem dla klasy BieliznaException i nikt nie zwróciłby na to uwagi. Dzieje się tak, ponieważ typ KoszulaException, choć ma większy zakres i może przechwytywać więcej klas (swoich klas potomnych), nie może jednak przechwytywać wyjątków klasy BieliznaException. A zatem wszystko jest w porządku.

Zagadka polimorficzna



```
try {  
    x.zrobCosRzykownego();  
} catch(AlfaEx a) {  
    // rozwiązanie problemu AlfaEx  
} catch(BetaEx b) {  
    // rozwiązanie problemu BetaEx  
} catch(GammaEx c) {  
    // rozwiązanie problemu GammaEx  
} catch(DeltaEx d) {  
    // rozwiązanie problemu DeltaEx  
}
```

Przyjmij, że przedstawiony obok fragment kodu jest *poprawny*. Twoim zadaniem jest narysowanie dwóch różnych diagramów klas, które będą poprawnie odzwierciedlać hierarchię klas wyjątków. Innymi słowy, jaka powinna być struktura dziedziczenia klas wyjątków, aby przedstawiony fragment kodu był poprawny?



Twoim zadaniem jest stworzenie dwóch *poprawnych* grup bloków try-catch (podobnych od kodu przedstawionego powyżej), które będą odpowiadać diagramowi klas przedstawionemu obok. Przyjmij, że metoda wywoływana w bloku try może zgłosić KAŻDY z wyjątków.

Kiedy nie chcesz obsługiwać wyjątku...

pomiń go

Jeśli nie chcesz obsługiwać wyjątku, możesz tego uniknąć, deklarując wyjątek.

Kiedy wywołujesz ryzykowną metodę, kompilator wymaga, abyś był tego świadomym. W większości przypadków oznacza to umieszczenie wywołania takiej metody wewnątrz bloku `try-catch`. Istnieje jednak inne rozwiązanie — wystarczy po prostu zrezygnować z obsługi wyjątku i przekazać go do metody, która wywołała *Twoją* metodę, aby ta ona go obsługiła.

Rozwiązanie to jest bardzo łatwe — sprowadza się do **zadeklarowania**, że metoda zgłasza wyjątek. Choć z technicznego punktu widzenia to nie *Ty* zgłaszasz wyjątek, to jednak nie ma to żadnego znaczenia. Tym razem to *Ty* pozwalasz na dalszą propagację wyjątku.

Jeśli jednak zrezygnujesz z obsługi wyjątku i nie umieścisz w swoim kodzie bloku `try-catch`, to co się stanie, gdy ryzykowna metoda (`zrobPranie()`) zgłosi wyjątek?

Kiedy metoda zgłosi wyjątek, jest ona bezzwłocznie usuwana ze stosu, a wyjątek jest ponownie zgłoszany i kierowany do kolejnej metody znajdującej się na wierzchołku stosu — czyli metody *wywołującej* Twoją metodę. Jeśli jednak także *ta* metoda woli *unikać* wyzwań, to również i *ona* nie obsługuje wyjątku i zostanie bezzwłocznie usunięta ze stosu, a wyjątek zostanie przekazany do kolejnej metody i tak dalej...

Kiedy to się skończy? Przekonasz się o tym już niedługo.

```
public void test() throws KoszmarException {
    // wywołaj ryzykowną metodę bez umieszczenia jej
    // w bloku try-catch
    pranie.zrobPranie();
}
```



W rzeczywistości to nie *Ty* zgłaszasz wyjątek, jednak, ponieważ nie umieścisz wywołania ryzykownej metody w bloku `try-catch`, zatem teraz Twoja metoda stała się „ryzykowna”, ponieważ w tej sytuacji to *kod*, który wywołuje *TWOJĄ* metodę, musi obsługiwać wyjątek.

Zrezygnowanie z obsługi wyjątku (poprzez jego zadeklarowanie) jedynie odsuwa w czasie to, co nieuniknione

Wcześniej czy później i tak ktoś musi obsłużyć wyjątek.

Co się jednak stanie, kiedy także metoda `main()`
zrezygnuje z obsługi wyjątku?

```
public class Pralka {  
    Pranie pranie = new Pranie();  
  
    public void test() throws OdziezException {  
        pranie.zrobPranie();  
    }  
  
    public static void main(String[] args) throws OdziezException {  
        Pralka pr = new Pralka();  
        pr.test();  
    }  
}
```

Obie metody rezygnują z obsługi wyjątku (deklarując go), nie ma zatem nikogo, kto mógłby ten wyjątek obstawić! Taki program można skompilować bez najmniejszego problemu.

- 1 Metoda `zrobPranie()` zgłasza wyjątek `OdziezException`.



Metoda `main()` wywołuje metodę `test()`
Metoda `test()` wywołuje metodę `zrobPranie()`
Metoda `zrobPranie()` zostaje uruchomiona i zgłasza wyjątek `OdziezException`

- 2 Metoda `test()` rezygnuje z obsługi wyjątku



Metoda `zrobPranie()` jest natychmiast usuwana ze stosu, a wyjątek zostaje ponownie zgłoszony i przekazany do metody `test()`.
Ale także w metodzie `test()` nie ma bloku `try-catch`, zatem...

- 3 Metoda `main()` zgłasza wyjątek `OdziezException`.



Metoda `test()` jest natychmiast usuwana ze stosu, a wyjątek jest zgłoszany ponownie i przekazywany... właśnie, do kogo? Gdzie? Została już jedynie wirtualna maszyna Javy, a ona sobie myśli: „Nie oczekuj bracie, że ja wyciągnę cię z kłopotów”.



W powyższym przykładzie ikonka koszulki reprezentuje wyjątek `OdziezException`. Tak, tak, wiemy... wolałbyś, żeby to były dżinsy.

Obsługuj lub deklaruj. Taka jest zasada.

A zatem poznaleś już oba sposoby spełnienia wymagań, jakie stawia przed nami kompilator w przypadku wywoływania ryzykownej metody (czyli metody zgłaszającej wyjątki).

1 OBSŁUGA

Umieść wywołanie ryzykownej metody w bloku try-catch

```
try {
    pranie.zrobPranie();
} catch(OdziezException oex) {
    // obsługa wyjątku
}
```

Lepiej, żeby zakres tego bloku catch był na tyle duży, aby były w nim obsługiwane wszystkie wyjątki `OdziezException`, które może zgłaszać metoda `zrobPranie()`. W przeciwnym razie kompilator będzie się „uskarżał”, że nie obsługujeesz wszystkich wyjątków.

2 ZADEKLAROWANIE (rezygnacja z obsługi)

Zadeklaruj, że TWOJA metoda zgłasza ten sam wyjątek, który może zgłosić wywoływana, ryzykowna metoda.

```
void test() throws OdziezException {
    pranie.zrobPranie();
}
```

Metoda `zrobPranie()` zgłasza wyjątek `OdziezException`; jednak, deklarując ten wyjątek, metoda `test()` może uniknąć jego obsługi. Blok try-catch nie jest konieczny.

Jednak w naszym przypadku oznacza to, że ktokolwiek wywoła metodę `test()`, będzie musiał postąpić zgodnie z prawem „obsłuż lub zadeklaruj”. Jeśli metoda `test()` zrezygnuje z obsługi wyjątku (deklarując go), a metoda `main()` wywoła `test()`, zatem to metoda `main()` musi przechwycić i obsługiwać wyjątek.

```
public class Pralka {
    Pranie pranie = new Pranie();
```

PROBLEM!!!

```
    public void test() throws OdziezException {
        pranie.zrobPranie();
    }
```

Teraz metoda `main()` nie chce się skompilować, a my uzyskujemy komunikat o „niezgłoszonym wyjątku”. Z punktu widzenia kompilatora, metoda `test()` zgłasza wyjątek.

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

```
        Pralka pr = new Pralka();
        pr.test();
```

Ponieważ metoda `test()` nie obsługuje wyjątku `OdziezException` zgłoszanego przez metodę `zrobPranie()`, zatem w metodzie `main()` wywołanie `test()` musi zostać umieszczone w bloku try-catch lub także metoda `main()` musi zadeklarować, że zgłasza wyjątek `OdziezException`.

Wróćmy do kodu naszej aplikacji muzycznej...

Zapewne zupełnie już zapomniałeś, że zaczeliśmy ten rozdział od przedstawienia fragmentu kodu wykorzystującego możliwości biblioteki JavaSound. Stworzyliśmy obiekt Sequencer, jednak nasz kod nie chciał się skompilować, gdyż metoda MidiSystem.getSequencer() deklarowała sprawdzany wyjątek (a konkretnie — MidiUnavailableException). Teraz jednak możemy rozwiązać nasz problem, umieszcając wywołanie tej metody wewnątrz bloku try-catch.

```
public void graj() {
    try {
        Sequencer sekvenser = MidiSystem.getSequencer();
        System.out.println("Mamy sekvenser!");

    } catch (MidiUnavailableException ex) { ←
        System.out.println("Masz problem");
    }
} // koniec graj
```

Teraz, po umieszczeniu wewnątrz bloku try-catch, wywołanie metody getSequencer() nie przysparza żadnych problemów.

Parametr bloku catch musi być „odpowiednim” wyjątkiem. Gdybyśmy użyli kodu catch(FileNotFoundException f), nie datoby się go skompilować, gdyż wyjątek MidiUnavailableException nie pasuje do wyjątku FileNotFoundException.
Pamiętaj, sam fakt umieszczenia w kodzie bloku catch nie wystarcza... trzeba przechwytywać w nim odpowiedni (czyli zgłoszany) wyjątek!

Reguły związane ze stosowaniem wyjątków

- 1 Nie można umieścić w kodzie bloków catch lub finally bez poprzedzenia ich blokiem try.**

```
void doDziela() {
    Test t = new Test();      TO JEST NIEDOZWOLONE!
    t.testfun();              Gdzie jest blok try?
    catch(TestException tex) { }
```

- 2 Nie można umieszczać żadnego kodu pomiędzy blokami try oraz catch.**

```
try {
    x.zrobCos();
}
int y = 43; ←
} catch(Exception ex) { }
```

TO JEST NIEDOZWOLONE!
Nie można umieszczać żadnego kodu pomiędzy blokami try i catch.

- 3 Po bloku try MUSI znaleźć się blok catch lub finally.**

```
try {
    x.zrobCos();
} finally {
    // zakończenie operacji
}
```

DOZWOLONE, gdyż istnieje blok finally, choć nie ma żadnego bloku catch. Jednak podanie samego bloku try byłoby błędem.

- 4 Jeśli zostanie użyty jedynie blok try oraz finally (bez bloku catch), to metoda i tak musi deklarować wyjątek.**

```
void doDziela() throws TestException {
    try {
        x.zrobCos();
    } finally { }
}
```

Kod, w którym podano sam blok try — bez bloku catch — nie realizuje zasad „obsłuż lub zadeklaruj”.

Kod od kuchni



Nie musisz robić tego sam, ale jeśli się zdecydujesz, będziesz mieć o wiele więcej zabawy.

Pozostała część tego rozdziału jest opcjonalna — zamiast pisać kod wszystkich aplikacji muzycznych, możesz wykorzystać „Kod gotowy do użycia”.

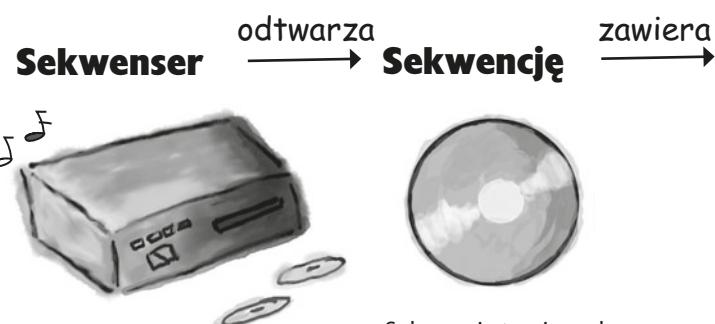
Jeśli jednak chciałbyś się dowiedzieć czegoś więcej na temat JavaSound, przewróć kartkę.

Generowanie dźwięków

Może pamiętasz, że na początku tego rozdziału analizowaliśmy dane MIDI zawierające instrukcje dotyczące tego, *co należy zagrać (i jak to coś należy zagrać)*. Napisaliśmy wtedy także, iż dane MIDI nie tworzą żadnych dźwięków, które moglibyśmy usłyszeć. Aby usłyszeć dźwięki z głośnika, trzeba przesłać dane MIDI do jakiegoś urządzenia MIDI, które przetworzy zawarte w nich instrukcje i zamieni je na dźwięki, wykorzystując w tym celu instrument sprzętowy lub „wirtualny” (syntezator programowy). W niniejszej książce będziemy używać wyłącznie syntezatora programowego, zatem poniżej opiszymy, jak to wszystko działa.

Potrzebujesz CZTERECH rzeczy:

- ① Czegoś, co odtwarza muzykę
- ② Muzyki, jaką należy odtworzyć... piosenki
- ③ Części sekwencji, która zawiera właściwe informacje
- ④ Właściwych informacji o muzyce: tonów, jakie należy zagrać, ich długości itd.

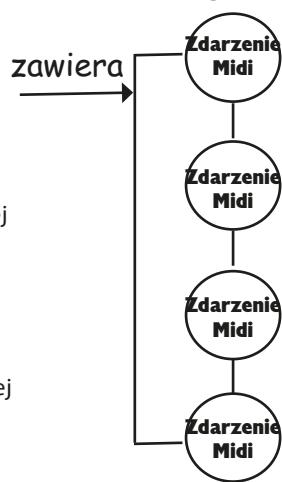


Sekwenser to urządzenie, które umożliwia odtworzenie muzyki. Możesz go sobie wyobrazić jako **odtwarzacz płyt kompaktowych**.

Sekwencja to piosenka — utwór muzyczny odtwarzany przez sekvenser. Analizując przykłady zamieszczone w tej książce, możesz ją sobie wyobrażać jako płytę kompaktową, jednak płytę, na której jest zapisana tylko jedna piosenka.

Analizując przykłady podane w tej książce, możesz sobie wyobrażać sekwencje jako płytę kompaktową, na której znajduje się tylko jeden utwór (jedna ścieżka). Informacje o tym, jak należy odtworzyć utwór, są zapisane na ścieżce, a ścieżka stanowi część sekwencji.

W przykładach przedstawionych w tej książce będziemy potrzebowali tylko jednej ścieżki, zatem wyobraź sobie, że posiadamy płytę kompaktową, na której jest zapisany tylko jeden utwór. Czyli pojedyncza ścieżka. Właśnie na tej ścieżce znajdują się wszystkie informacje o piosenke (czyli dane MIDI).



Zdarzenia MIDI to komunikaty, które może zrozumieć sekvenser. Gdyby takie zdarzenie mówiło po polsku, mogłoby nakazać sekvenserowi: „W tym momencie zagraj wysokie C, zagraj je mocno i szybko; odtwarzaj je przez podany czas”.

Zdarzenie MIDI może także zażądać, na przykład, zmiany instrumentu na flet.

Będziesz musiał wykonać PIĘĆ czynności:

- ① Pobrać sekvenser (obiekt Sequencer) i otworzyć go:

```
Sequencer sekvenser = MidiSystem.getSequencer();  
sekvenser.open();
```

- ② Utworzyć nowy obiekt Sequence:

```
Sequence sekw = new Sequence(tempo,4)
```

- ③ Pobrać z sekwencji (obiektu Sequence) nową ścieżkę (obiekt Track):

```
Track t = sekw.createTrack();
```

- ④ Wypełnić ścieżkę obiektami MidiEvent
i przekazać sekwencję do sekvensera:

```
t.add(mojeZdarzenieMidi1);  
sekvenser.setSequence(sekw);
```



sekvenser.start();

Twój pierwszy odtwarzacz muzyki

Wpisz poniższy kod i uruchom go. Usłyszysz dźwięk, jak gdyby ktoś zagrał na fortepianie pojedynczą nutę.

```
import javax.sound.midi.*; ← Nie zapomnij zainportować pakietu Midi.  
public class MiniMiniMuzaApk { // pierwsza aplikacja  
  
    public static void main(String[] args) {  
        MiniMiniMuzaApk mini = new MiniMiniMuzaApk();  
        mini.graj();  
    } // koniec main  
  
    public void graj() {  
        try {  
  
            ① Sequencer sekvenser = MidiSystem.getSequencer();  
            sekvenser.open(); ← Pobieramy obiekt Sequencer  
                               i otwieramy go (abyśmy mogli go  
                               używać, gdyż po pobraniu sekvenser  
                               nie jest otwarty).  
  
            ② Sequence sekw = new Sequence(Sequence.PPQ, 4);  
  
            ③ Track sciezka = sekw.createTrack(); ← Nie zwracaj na razie uwagi na argumenty  
                                            przekazywane do konstruktora ścieżki  
                                            — obiektu Sequence. Po prostu skopiuj te,  
                                            które podaliśmy (wyobraź sobie, że są to  
„Argumenty gotowe do użycia”).  
  
            ④ ShortMessage a = new ShortMessage();  
            a.setMessage(144, 1, 44, 100);  
            MidiEvent nutaP = new MidiEvent(a, 1);  
            sciezka.add(nutaP);  
  
            ShortMessage b = new ShortMessage();  
            b.setMessage(128, 1, 44, 100);  
            MidiEvent nutaK = new MidiEvent(b, 16);  
            sciezka.add(nutaK);  
  
            sekvenser.setSequence(sekw); ← Poproś obiekt Sequence o zwrócenie ścieżki  
                                         — obiektu Track. Pamiętaj, że obiekt Track  
                                         istnieje wewnątrz obiektu Sequence, a dane  
                                         MIDI — wewnątrz obiektu Track.  
  
            sekvenser.start(); ← Umieść w obiekcie Track jakieś zdarzenia  
                               MIDI (obiekty MidiEvent). Ta część  
                               naszego przykładu została przygotowana  
                               wcześniej. Jedyna rzecz, na jaką możesz  
                               zwrócić uwagę, to argumenty metody  
                               setMessage() oraz konstruktora klasy  
                               MidiEvent. Przyjrzymy się im dokładniej  
                               na następnej stronie.  
  
        } catch (Exception ex) {ex.printStackTrace();}  
    } // koniec graj  
} // koniec klasy
```



Tworzenie obiektów MidiEvent (danych piosenki)

Zdarzenia MidiEvent to instrukcje dotyczące fragmentów piosenki. Serię takich zdarzeń można porównać do zapisu nutowego lub zapisu klawiszowego. Nas najbardziej będą obchodzić zdarzenia opisujące, *co należy zrobić* oraz *w jakim momencie to coś należy zrobić*. Momenty mają znaczenie, gdyż odpowiednia synchronizacja zdarzeń jest w muzyce niezmiernie istotna. Ta nuta następuje po tamtej i tak dalej. Ponieważ zdarzenia MidiEvent są tak szczegółowe, musimy zatem określić, w jakim momencie należy zacząć grać daną nutę (zdarzenie NOTE ON) oraz kiedy ją zakończyć (zdarzenie NOTE OFF). Możesz sobie wyobrazić, że zgłoszenie zdarzenia „zakończ granie dźwięku G” (NOTE OFF) przed zgłoszeniem zdarzenia „rozłącz granie dźwięku G” (NOTE ON) nie dałoby pożądanych rezultatów.

Instrukcje MIDI są w rzeczywistości zapisywane w obiekcie komunikatu — Message — a każde zdarzenie MidiEvent stanowi kombinację takiego komunikatu oraz określenia czasu, w jakim ten komunikat należy „wykonać”. Innymi słowy, komunikat może stwierdzać „zaczni grać G”, a zdarzenie MidiEvent: „wykonaj ten komunikat w czwartym takcie”.

A zatem zawsze będzie nam potrzebny obiekt Message oraz MidiEvent.

Obiekt Message określa, *co należy zrobić*, a obiekt MidiEvent — *kiedy*.

1 Utworzenie obiektu komunikatu:

```
ShortMessage a = new ShortMessage();
```

Ten komunikat informuje: „zaczni grać nutę 44” (innym numerem nut przyjrzymy się na następnej stronie).

2 Umieszczenie instrukcji w komunikacie:

```
a.setMessage(144, 1, 44, 100);
```

3 Utworzenie zdarzenia MidiEvent przy wykorzystaniu komunikatu:

```
MidiEvent nutaP = new MidiEvent(a, 1);
```

Instrukcje zostały zapisane w komunikacie, jednak zdarzenie MidiEvent dodaje do nich określenie momentu, kiedy te instrukcje należy wykonać. Przedstawione zdarzenie informuje, że instrukcję „a” należy wykonać w pierwszym takcie (takcie o numerze 1).

4 Dodanie zdarzenia do ścieżki:

```
sciezka.add(nutaP);
```

Wszystkie zdarzenia MidiEvent są przechowywane na ścieżce. Sekwencja porządkuje zdarzenia na podstawie czasu, w którym mają zostać wykonane, a następnie sekwenser odtwarza je w określonej kolejności. W tej samej chwili może zachodzić dowolnie wiele zdarzeń. Na przykład, mógłbyś chcieć, aby jednocześnie były grane dwie nuty, a nawet, żeby w tej samej chwili dwa instrumenty graly dwa różne dźwięki.

Zdarzenie MidiEvent określa, *co* oraz *kiedy* należy zrobić.

Każda instrukcja musi zawierać określenie *momentu*, kiedy należy ją wykonać.

Innymi słowy
— w którym *takcie* ma nastąpić dane zdarzenie.

Komunikat MIDI — serce zdarzenia MidiEvent

Komunikat MIDI zawiera tę część zdarzenia, która informuje, *co* należy zrobić. Jest to właściwa instrukcja, którą ma wykonać sekvenser. Pierwszym argumentem tej instrukcji jest typ komunikatu. Na przykład typ o numerze 144 oznacza „POCZĄTEK NUTY” (NOTE ON). Jednak aby wykonać ten komunikat, sekvenser musi uzyskać kilka dodatkowych informacji. Wyobraź sobie, że sekvenser oznajmia: „No dobrze... zagram tą nutę, ale *jakiego kanalu* mam użyć? Innymi słowy, czy chcesz, żeby to był dźwięk bębna czy fortepianu? A w ogóle, to *jaką nutę* mam zagrać? C? Dis? A skoro już o tym mowa, to *jak długo* ma trwać dźwięk?”.

Aby utworzyć komunikat MIDI, musisz stworzyć obiekt klasy `ShortMessage`, a następnie wywołać metodę `setMessage()`, przekazując do niej cztery argumenty. Pamiętaj jednak, że komunikat informuje jedynie o tym, co należy zrobić; wciąż zatem musisz umieścić komunikat w zdarzeniu i określić, *kiedy* należy go wykonać.

Anatomia komunikatu

Pierwszy argument wywołania metody `setMessage()` zawsze określa „typ” komunikatu, natomiast znaczenie kolejnych trzech argumentów jest różne dla komunikatów różnych typów.

```
a.setMessage(144, 1, 44, 100);
```

typ komunikatu
kanal nuta, która należy zagrać szybkość

Znaczenie trzech ostatnich argumentów zmienia się w zależności od typu komunikatu. To jest komunikat NOTE ON, a zatem pozostałe trzy argumenty określają informacje, jakich sekvenser potrzebuje, aby zagrać nutę.

1 Typ komunikatu.

Numer 144 oznacza zdarzenie NOTE ON



Numer 128 oznacza zdarzenie NOTE OFF



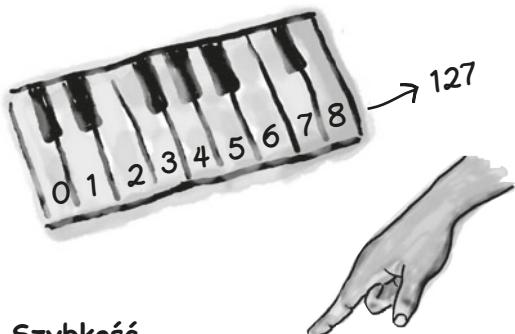
Obiekt `ShortMessage` określa, co należy zrobić, a *MidiEvent* — kiedy.

2 Kanał

Kanał możesz sobie wyobrazić jako jednego z muzyków wchodzących w skład zespołu. Kanał 1. to pierwszy muzyk (grający na instrumentach klawiszowych), kanał 9. to perkusista i tak dalej.

3 Grana nuta

Może przyjmować numery od 0 do 127 odpowiadające coraz to wyższym dźwiękom.



4 Szybkość

Jak szybko i mocno nacisnęłaś klawisz? Oznacza „muśnięcie”, którego pewnie nawet byś nie usłyszał, ale 100 to doskonała wartość domyślna.

Zmiana komunikatu

Teraz, kiedy już wiesz, jakie informacje tworzą komunikat MIDI, możesz eksperymentować. Możesz zmieniać nutę, jaka będzie grana, czas jej trwania, dodawać więcej nut, a nawet zmieniać instrumenty.

1 Zmiana nuty.

W komunikatach reprezentujących początek i koniec nuty spróbuj podawać liczby z zakresu do 0 do 127.

```
a.setMessage(144, 1, 20, 100);
```



2 Zmiana długości nuty.

Zmień zdarzenie (nie komunikat) zakończenia nuty, tak aby zostało wykonane we wcześniejszym lub późniejszym takcie.

```
b.setMessage(128, 1, 44, 100);
```

```
MidiEvent nutaK = new MidiEvent(b, 3);
```



3 Zmiana instrumentu.

PRZED komunikatem rozpoczętym granie nuty dodaj nowy komunikat, który zmienia instrument używany na kanale pierwszym na coś innego niż fortepian (domyślnie używany na tym kanale). Komunikat zmiany instrumentu ma numer 192, a nowy instrument, jaki ma być używany, jest określany przez trzeci argument wywołania metody `setMessage()` (może on przyjmować wartości z zakresu od 0 do 127).

```
pierwszy.setMessage(192, 1, 102, 0);
```

komunikat zmiany instrumentu
 na kanale 1. (pierwszy muzyk)
 na instrument o numerze 102

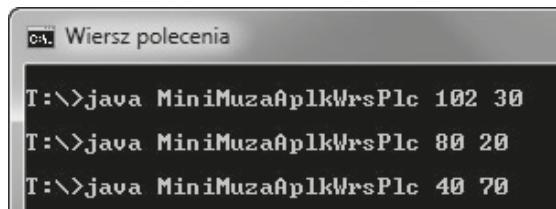


Wersja 2. Eksperymenty muzyczne przy wykorzystaniu argumentów przekazywanych z wiersza poleceń

Przedstawiona wcześniej wersja aplikacji odtwarza tylko jedną nutę, jednak można wykorzystać argumenty przekazywane w wierszu poleceń, aby zmieniać instrument oraz nutę. Użyj poniższego programu do przeprowadzania eksperymentów, podając w wierszu wywołania dwie liczby całkowite z zakresu od 0 do 127. Pierwsza liczba określa instrument, a druga grą nutę.

```
import javax.sound.midi.*;  
  
public class MiniMuzaAplkWrsPlc { // pierwsza wersja aplikacji  
  
    public static void main(String[] args) {  
        MiniMuzaAplkWrsPlc mini = new MiniMuzaAplkWrsPlc();  
        if (args.length < 2) {  
            System.out.println("Nie zapomnij podać argumentów określających instrument i nutę");  
        } else {  
            int instrument = Integer.parseInt(args[0]);  
            int nuta = Integer.parseInt(args[1]);  
            mini.graj(instrument, nuta);  
        }  
    } // koniec main  
  
    public void graj(int instrument, int nuta) {  
  
        try {  
  
            Sequencer sekvenser = MidiSystem.getSequencer();  
            sekvenser.open();  
  
            Sequence sekw = new Sequence(Sequence.PPQ, 4);  
            Track sciezka = sekw.createTrack();  
  
            MidiEvent zdarzenie = null;  
  
            ShortMessage pierwszy = new ShortMessage();  
            pierwszy.setMessage(192, 1, instrument, 0);  
            MidiEvent zmienInstrument = new MidiEvent(pierwszy, 1);  
            sciezka.add(zmienInstrument);  
  
            ShortMessage a = new ShortMessage();  
            a.setMessage(144, 1, nuta, 100);  
            MidiEvent nutaP = new MidiEvent(a, 1);  
            sciezka.add(nutaP);  
  
            ShortMessage b = new ShortMessage();  
            b.setMessage(128, 1, nuta, 100);  
            MidiEvent nutaK = new MidiEvent(b, 16);  
            sciezka.add(nutaK);  
            sekvenser.setSequence(sekw);  
            sekvenser.start();  
  
        } catch (Exception ex) {ex.printStackTrace();}  
    } // koniec graj  
} // koniec klasy
```

Uruchom program, podając w wierszu poleceń dwie liczby z zakresu od 0 do 127.

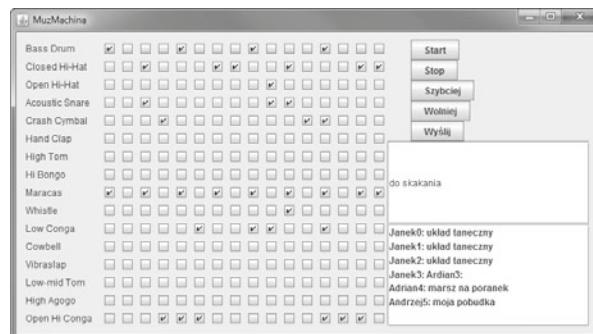


```
T:>>java MiniMuzaAplkWrsPlc 102 30  
T:>>java MiniMuzaAplkWrsPlc 80 20  
T:>>java MiniMuzaAplkWrsPlc 40 70
```

Jaki cel staraliśmy się osiągnąć w pozostałych przykładach przedstawionych w ramach „Kodu od kuchni”?

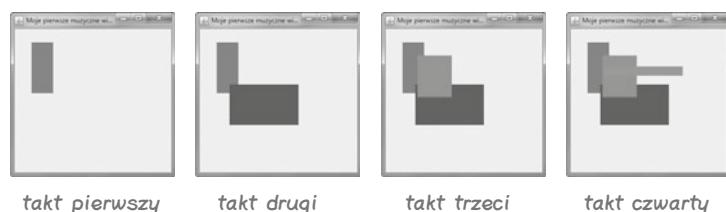
Rozdział 15. Cel ostateczny

Kiedy zakończymy, będziemy dysponowali działającym programem MuzMachina, który jednocześnie będzie pełnił funkcję klienta Pogawędka perkusyjnych. Aby go stworzyć, będziemy musieli poznać zasady tworzenia graficznego interfejsu użytkownika (wraz z obsługą zdarzeń), nauczyć się obsługi wejścia-wyjścia, komunikacji sieciowej oraz wątków. Wszystkie te umiejętności zdobędziemy w kolejnych trzech rozdziałach.



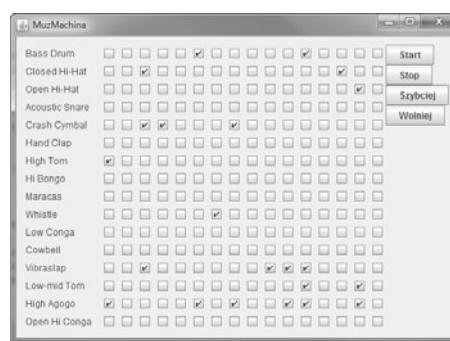
Rozdział 12. Zdarzenia MIDI

Ten „Kod od kuchni” pozwoli nam stworzyć proste „muzyczne wideo” (choć to trochę zbyt szumnie powiedziane), które wyświetla losowo generowane prostokąty w takt muzyki MIDI. Dowiemy się, jak tworzyć i odtwarzać wiele różnych zdarzeń MIDI (a nie tylko kilka, o których była mowa w tym rozdziale).



Rozdział 13. Samodzielna aplikacja MuzMachina

W tym rozdziale stworzymy właściwą aplikację MuzMachina wyposażoną w graficzny interfejs użytkownika i wszystkie inne „wodotryski”. Jednak jej możliwości będą ograniczone — jak tylko zmienisz wzorzec, poprzednia kompozycja zostanie utracona. Nasza aplikacja nie dysponuje opcją „zapisz i odczytaj”, nie potrafi komunikować się za pośrednictwem sieci. (Niemniej jednak z powodzeniem można jej używać do rozwijania swoich talentów perkusyjnych).



Rozdział 14. Zapis i odczytywanie

Stworzyłeś doskonałą kompozycję i teraz chciałbyś ją zapisać w pliku, a w przyszłości — odczytać i ponownie odtworzyć. Te możliwości zbliżają nas ku ostatecznej wersji aplikacji (przedstawionej w rozdziale 15.), w której kompozycja nie jest zapisywana w pliku, lecz przesyłana poprzez sieć do specjalnego serwera pogawędka muzycznych.



Ćwiczenia: Prawda czy fałsz



W tym rozdziale poznaliśmy wspaniały świat wyjątków. Twoim zadaniem jest określenie, czy poniższe stwierdzenia są prawdziwe czy też fałszywe.

👉 PRAWDA CZY FAŁSZ 👈

1. Po bloku `try` trzeba umieścić blok `catch i` blok `finally`.
2. Tworząc metodę, która może spowodować zgłoszenie wyjątku sprawdzanego przez kompilator, „ryzykowny” kod `trzeba` umieścić wewnątrz bloku `try–catch`.
3. W blokach `catch` można wykorzystywać mechanizm polimorfizmu.
4. Jedynie wyjątki sprawdzane przez kompilator mogą być przechwytywane.
5. Jeśli zdefiniujesz bloki `try` oraz `catch`, blok `finally` stanie się opcjonalny.
6. Jeśli zdefiniujesz blok `try`, to możesz do niego dodać blok `catch`, blok `finally` lub oba te bloki jednocześnie.
7. Tworząc metodę, która deklaruje możliwość zgłaszania wyjątków sprawdzanych, kod, który może je zgłosić, trzeba umieścić w bloku `try–catch`.
8. Metoda `main()` programu musi obsługiwać wszystkie przekazane do niej wyjątki, które nie zostały obsłużone wcześniej.
9. Jednemu blokowi `try` może towarzyszyć wiele różnych bloków `catch`.
10. Metoda może zgłaszać tylko jeden typ wyjątków.
11. Blok `finally` zostanie wykonany niezależnie od tego, czy został zgłoszony jakiś wyjątek czy nie.
12. W kodzie można umieścić blok `finally`, nawet jeśli nie zostanie on poprzedzony blokiem `try`.
13. Blok `try` może istnieć samodzielnie — bez towarzyszących mu bloków `catch` oraz `finally`.
14. Obsługiwanie wyjątku jest czasami określane mianem jego „unikania”.
15. Kolejność zapisu bloków `catch` nigdy nie ma znaczenia.
16. Metoda zawierające bloki `try` oraz `finally` może, opcjonalnie, deklarować wyjątek.
17. Wyjątki zgłasiane podczas wykonywania programu muszą być *obsługiwane bądź deklarowane*.



Ćwiczenie



Magnesiki z kodem

Działający program Javy został podzielony na fragmenty, zapisany na małych magnesach, które przyczepiono do lodówki. Czy jesteś w stanie złożyć go z powrotem w jedną całość, tak aby wygenerował przedstawione poniżej wyniki? Niektóre nawiasy klamrowe spadły na podłogę i były zbyt małe, aby można je było podnieść, dlatego, w razie potrzeby, możesz je dodawać dowolnie!

System.out.print("p");

try {

System.out.print("j");

System.out.println("a");

} finally {

System.out.print("n");

System.out.print("i");

class MojEx extends Exception { }

public class ExTester {

System.out.print("a");

if ("tak".equals(t)) {

System.out.print("a");

throw new MojEx();

} catch (MojEx e) {

public static void main(String[] args) {
String test = args[0];static void zaryzykuj(String t) throws MojEx {
System.out.print("i");

Wiersz poleceń

T:\>java ExTester tak
pianaT:\>java ExTester nie
pijana



Rozwiązańia ćwiczeń

PRAWDA CZY FAŁSZ

1. Fałsz, może wystąpić dowolny z tych bloków lub oba jednocześnie.
2. Fałsz, można także zadeklarować wyjątek.
3. Prawda,
4. Fałsz, wyjątki `RuntimeException` także można przechwytywać.
5. Prawda.
6. Prawda, można dodać oba bloki.
7. Fałsz, wystarczy deklaracja.
8. Fałsz, jeśli jednak metoda `main()` nie obsługuje tych wyjątków, może to doprowadzić do zatrzymania wirtualnej maszyny Javy.
9. Prawda.
10. Fałsz.
11. Prawda, często jest on używany do zakończenia częściowo wykonanych zadań.
12. Fałsz.
13. Fałsz.
14. Fałsz, „unikanie” jest równoznaczne z zadeklarowaniem wyjątku.
15. Fałsz, wyjątki o najszerszym zakresie muszą być przechwytywane jako ostatnie.
16. Fałsz, jeśli nie stworzono bloku `catch`, to metoda *musi* deklarować wyjątek.
17. Fałsz.

Magnesiki z kodem

```
class MojEx extends Exception { }

public class ExTester {

    public static void main(String[] args) {
        String test = args[0];
        try {
            System.out.print("p");
            zaryzykuj(test);
            System.out.print("a");
        } catch (MojEx e) {
            System.out.print("a");
        } finally {
            System.out.print("n");
        }
        System.out.println("a");
    }

    static void zaryzykuj(String t) throws MojEx {
        System.out.print("i");
        if ("tak".equals(t)) {
            throw new MojEx();
        }
        System.out.print("j");
    }
}
```

The screenshot shows a terminal window titled "Wiersz polecenia" (Terminal). It displays the following command and its output:

```
T:\>java ExTester tak
piana
T:\>java ExTester nie
pijana
```

12. Tworzenie graficznego interfejsu użytkownika

Historia bardzo graficzna

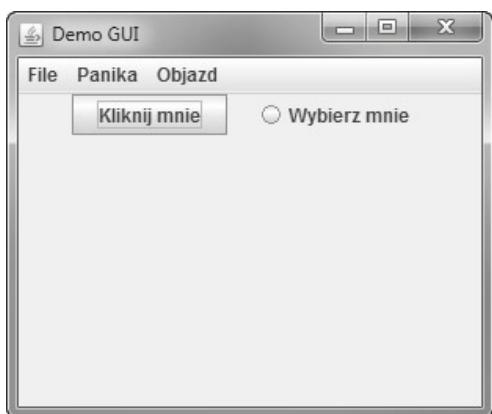


Pogódź się z faktem, że potrzebujesz graficznego interfejsu użytkownika. Tworząc programy przeznaczone dla innych osób, musisz wyposażyć je w graficzny interfejs użytkownika. Tworząc programy na własny użytek, będziesz chciał wyposażyć je w graficzny interfejs użytkownika. Nawet jeśli wierzysz, że całe życie spędzasz na pisaniu programów wykonywanych po stronie serwera, gdzie funkcję interfejsu użytkownika pełnią strony WWW, to i tak wcześniej czy później będziesz musiał napisać jakiś program narzędziowy i zapewne zechcesz wyposażyć go w graficzny interfejs użytkownika. Oczywiście programy obsługiwane z poziomu wiersza poleceń są w stylu retro, jednak w złym znaczeniu tego słowa. Takie programy są słabe, nieelastyczne i nieprzyjazne użytkownikowi. Ten oraz kolejny rozdział książki poświęcimy pracy nad graficznym interfejsem użytkownika, ucząc się przy okazji kluczowych cech Javy, takich jak: **obsługa zdarzeń** oraz **klasy wewnętrzne**. W tym rozdziale wyświetlimy przycisk i sprawimy, że jego kliknięcie spowoduje wykonanie pewnych czynności. Oprócz tego narysujemy coś na ekranie, wyświetlimy na nim obrazek zapisany w formacie JPEG, a nawet stworzymy prostą animację.

Wszystko zaczyna się od okna

JFrame to obiekt reprezentujący okno wyświetlane na ekranie. To właśnie w nim umieszczane są wszystkie komponenty, takie jak przyciski, pola wyboru, pola tekstowe i tak dalej. Okno może także zawierać pasek menu z opcjami. No i oczywiście ma także te wszystkie niewielkie ikony o wyglądzie dostosowanym do używanego systemu operacyjnego i służące do minimalizacji, maksymalizacji i zamykania okna.

Okna tworzone przy użyciu obiektu JFrame wyglądają inaczej w zależności od używanego systemu operacyjnego. Na przykład, tak wygląda okno w systemie Windows:



Umieszczaj komponenty w oknie

Po utworzeniu okna można umieszczać na nim komponenty, dodając je do obiektu JFrame. W bibliotece Swing istnieją dziesiątki takich elementów graficznego interfejsu użytkownika, a informacje o nich można znaleźć w dokumentacji pakietu javax.swing. Oto kilka najczęściej stosowanych komponentów: JButton, JRadioButton, JCheckBox, JLabel, JList, JScrollPane, JSlider, JTextArea, JTextField oraz JTable. Większość z nich jest naprawdę prosta w użyciu, lecz niektóre (na przykład JTable) mogą być nieco złożone.



„Jeśli zobaczę jeszcze jedną aplikację obsługiwana z poziomu wiersza poleceń, zostaniesz zwolniony”.

JFrame z menu oraz dwoma komponentami — przyciskiem i przyciskiem opcji.

Tworzenie graficznego interfejsu użytkownika jest proste:

- 1 Utwórz ramkę (obiekt JFrame).
`JFrame ramka = new JFrame();`
- 2 Utwórz komponent (przycisk, pole tekstowe, itp.).
`JButton przycisk = new JButton("Kliknij mnie");`
- 3 Dodaj komponent do ramki.
`ramka.getContentPane().add(przycisk);`

Komponenty nie są dodawane bezpośrednio do ramki. Ramkę można sobie wyobrazić jako cienki pasek wokół okna, natomiast wszystkie komponenty dodawane są do panelu znajdującego się wewnątrz okna.
- 4 Wyświetl ramkę (określ jej wielkość i spraw, że będzie widoczna).
`ramka.setSize(300,300);
ramka.setVisible(true);`

Twój pierwszy interfejs graficzny — przycisk w ramce

```
import javax.swing.*; ← Nie zapomnij zainportować pakietu Swing.
```

```
public class ProstyGUI1 {
    public static void main(String[] args) {
        JFrame ramka = new JFrame(); ← Utwórz ramkę i przycisk.
        JButton przycisk = new JButton("Kliknij mnie"); (W wywołaniu konstruktora przycisku można przekazać tekst, który ma być na nim wyświetlony).
        ramka.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); ← Ten wiersz kodu zapewnia, że program zostanie zakończony bezpośrednio po zamknięciu okna.
        ramka.getContentPane().add(przycisk);
        ramka.setSize(300,300); ← Dodaj przycisk do panelu zawartości okna.
        ramka.setVisible(true); ← Określ wielkość ramki w pikselach.
    }
}
```

W końcu określ, że ramka ma być widoczna!! (Jeśli zapomnisz o tym wierszu kodu, to po uruchomieniu programu na ekranie nic się nie pojawi).

Zobaczmy, co się stanie, gdy uruchomimy ten program:

```
>java ProstyGUI1
```



O rany! To naprawdę DUŻY przycisk.

Przycisk zajął cały dostępny obszar ramki. W dalszej części rozdziału dowiemy się, jak kontrolować położenie (oraz wielkość) przycisku w ramce.

Ale gdy klikam przycisk, nic się nie dzieje...

To nie do końca prawda. Gdy klikasz przycisk, zmienia się jego wygląd — przycisk staje się „wciśnięty” (sposób prezentacji „wciśniętego” przycisku zależy od używanego sposobu prezentacji (określano jako „look and feel”), niemniej jednak przycisk zawsze w jakiś sposób pokaże, że został naciśnięty).

Prawdziwe pytanie brzmi: „W jaki sposób wykonać jakąś operację w odpowiedzi na kliknięcie przycisku?”.

Potrzebujemy teraz dwóch rzeczy:

- ① **Metody**, która zostanie wywołana, gdy użytkownik kliknie przycisk (czyli czynności, jaka ma zostać wykonana w wyniku kliknięcia).
- ② **Sposobu**, który pozwoli nam się dowiedzieć, kiedy wywołać tę metodę.



Chcemy wiedzieć, kiedy użytkownik kliknie.

Chcemy wiedzieć o zdarzeniu „kliknięcia przycisku”.

Nie istnieja głupie pytania

P: Czy przycisk będzie wyglądać jak zwyczajny przycisk systemu MacOS, gdy uruchomimy ten program na Macintoszu?

O: Jeśli tego sobie będziesz życzyć. Możesz wybierać pomiędzy kilkoma różnymi „sposobami prezentacji” — specjalnymi klasami biblioteki Swing, które kontrolują, jak ma wyglądać interfejs użytkownika. W większości przypadków będziesz mieć do wyboru przynajmniej dwa sposoby prezentacji — standardowy sposób prezentacji Javy (znany pod nazwą **Metal**) oraz sposób charakterystyczny dla używanego systemu operacyjnego. Tworząc zrzuty prezentowane w niniejszej książce wykorzystywaliśmy sposoby prezentacji **Windows** lub **Metal**.

P: Czy mogę sprawić, że program zawsze będzie wyglądać jak aplikacja systemu Windows, nawet gdy zostanie uruchomiony na komputerze Macintosh?

O: Nie. Nie wszystkie sposoby prezentacji są dostępne we wszystkich systemach operacyjnych. Jeśli chcesz się zabezpieczyć, możesz jawnie używać sposobu prezentacji Metal, dzięki czemu wygląd interfejsu graficznego zawsze będzie taki sam, niezależnie od używanego systemu operacyjnego; ewentualnie możesz nie określić sposobu prezentacji i wykorzystać ustawienia domyślne.

P: Słyszałem, że biblioteka Swing jest strasznie wolna i że nikt jej nie używa.

O: W przeszłości faktycznie tak było, jednak ta sytuacja uległa zmianie. Na słabzych komputerach możesz zauważyc pewne spowolnienie działania programu wynikające z użycia biblioteki Swing. Jednak jeśli dysponujesz nowoczesnym komputerem i wykorzystasz Javę 1.3 lub nowszą, to pewnie nawet nie zauważysz różnic w szybkości działania interfejsu graficznego Javy oraz rodzimego interfejsu graficznego używanego przez system operacyjny.

Przechwytywanie zdarzeń generowanych przez działania użytkownika

Wyobraź sobie, że w momencie kliknięcia przycisku chciałbyś zmienić wyświetlany na nim tekst „Kliknij mnie” na „Zostałem kliknięty”. W pierwszej kolejności możemy napisać metodę, która zmieni tekst widoczny na przycisku (pobieżne przejrzenie dokumentacji API pozwoli Ci określić, jakiej metody należy w tym celu użyć):

```
public void zmienTekst() {
    przycisk.setText("Zostałem kliknięty");
}
```

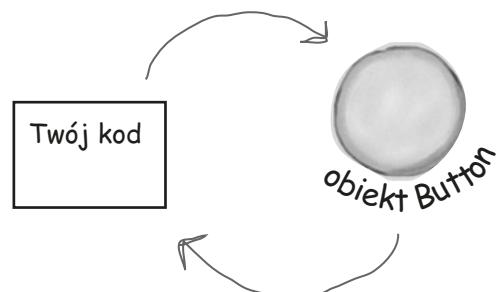
Ale co teraz? Jak mamy się *dowiedzieć*, kiedy tę metodę należy wywołać? **Jak mamy się dowiedzieć, że przycisk został kliknięty?**

W Javie proces zdobywania i obsługi zdarzeń generowanych przez poczynania użytkownika nazywany jest *obsługą zdarzeń*. Istnieje wiele różnych typów zdarzeń, choć większość z nich jest związana z operacjami wykonywanymi przez użytkownika na elementach graficznego interfejsu użytkownika. Kiedy użytkownik kliknie przycisk, tworzone jest zdarzenie reprezentujące tę czynność. „Użytkownik chce wykonać operację skojarzoną z tym przyciskiem”. Jeśli klikniętym przyciskiem jest „Odtwarzaj wolno”, to użytkownik chce, aby została wykonana czynność „wolnego odtwarzania”. Jeśli jest to przycisk „Wyślij” w programie do prowadzenia pogawędek internetowych, to użytkownik chce, żeby została wykonana czynność wysłania wiadomości. A zatem najprostsze zdarzenie jest generowane wtedy, gdy użytkownik kliknie przycisk, informując tym samym, że chce, aby została wykonana pewna czynność.

W przypadku przycisków nie będą Cię zazwyczaj interesować jakiekolwiek „pośrednie” zdarzenia, takie jak „przycisk został wciśnięty” lub „zwolniono przycisk”. Chcesz przekazać przyciskowi następującą instrukcję: „Nie interesuje mnie, co użytkownik robi z przyciskiem, jak długo trzyma wskaźnik myszy w jego obszarze, ile razy zmieni zdanie i przesunie wskaźnik myszy gdzie indziej i tak dalej. **Po prostu powiedz mi, kiedy użytkownik chce przystąpić do działania!** Innymi słowy, nie informuj mnie o niczym, chyba że użytkownik kliknie przycisk sygnalizując, że chce zrobić to, co przycisk potrafi!”.

Po pierwsze, przycisk musi wiedzieć, że jesteśmy nim zainteresowani.

1 Ej... Przycisku... Interesuje mnie to, co się z tobą dzieje.



2 Użytkownik mnie kliknął!

Po drugie, przycisk musi dysponować jakimś sposobem wywołania naszego kodu, kiedy zajdzie zdarzenie kliknięcia.

WYSIŁ SZARE KOMÓRKI

- 1) W jaki sposób mógłbyś poinformować obiekt przycisku, że interesujesz się jego zdarzeniami, że jesteś uważnym odbiorcą?
- 2) W jaki sposób przycisk wywoła Twój kod? Załóż, że nie masz możliwości przekazania do przycisku nazwy swojej unikalnej metody (zmienTekst()). W jaki zatem inny sposób możemy przekonać przycisk, że dysponujemy konkretną metodą, którą będzie mógł wywołać w momencie zajścia zdarzenia? (Podpowiedź: pomyśl o klasie Zwierze).

Odbiorcy zdarzeń

Jeśli interesują Cię zdarzenia generowane przez przycisk, **zaimplementuj interfejs**, który informuje: „**Chcę odbierać twoje zdarzenia**”.

Interfejs odbiorcy jest „mostem” łączącym **odbiorcę** (czyli Ciebie) oraz **źródło zdarzeń** (w tym przypadku — przycisk).

Źródłami zdarzeń są komponenty graficznego interfejsu użytkownika należące do biblioteki Swing. Z punktu widzenia Javy źródłem zdarzeń są obiekty, które mogą zamieniać operacje wykonywane przez użytkownika (kliknięcie przycisku myszy, naciśnięcie klawisza, zamknięcie okna) na zdarzenia. Niemal jak wszystko w Javie także i zdarzenia są reprezentowane jako obiekty. Obiekty pewnych klas. Jeśli zajrzesz do dokumentacji pakietu `java.awt.event`, znajdziesz tam sporo klas zdarzeń (bez trudu się zorientujesz, jakie to klasy, gdyż w ich nazwach zawsze pojawia się słowo **Event**, ang. zdarzenie). Zauważysz zapewne klasy `MouseEvent`, `KeyEvent`, `WindowEvent` oraz kilka innych.

W chwili, gdy użytkownik wykona jakiś znaczącą czynność (na przykład kliknie przycisk), **źródło** zdarzeń (takie jak przycisk) utworzy **obiekt zdarzenia**. W większości przypadków Twój kod będzie raczej *odbierać*, a nie *tworzyć* zdarzenia (dotyczy to całego kodu przedstawionego w niniejszej książce). Innymi słowy, w większości przypadków Twój kod będzie działać jako *odbiorca* zdarzeń, a nie ich *źródło*.

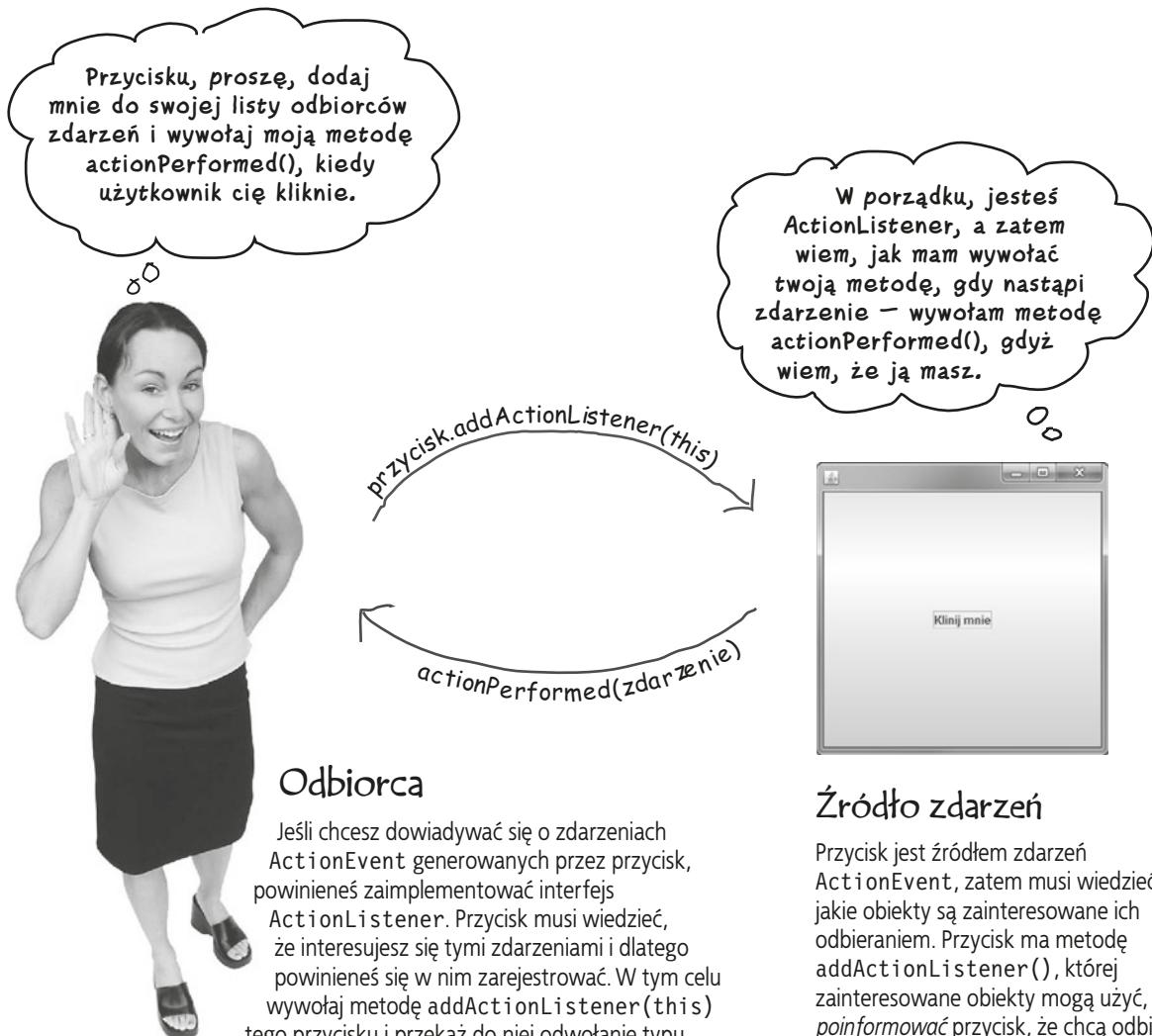
Każdy typ zdarzeń ma odpowiadający mu interfejs odbiorcy. Jeśli chcesz przechwytywać zdarzenia `MouseEvent`, powinieneś zaimplementować interfejs `MouseListener`. Chcesz obsługiwać zdarzenia `WindowEvent`? Zaimplementuj interfejs `WindowListener`. Na pewno już rozumiesz, o co chodzi. I nie zapominaj o regułach dotyczących interfejsów — aby zaimplementować interfejs, musisz zadeklarować, że go implementujesz (`class Pies implements ZwierzakDomowy`), a to oznacza, że musisz *stworzyć implementacje wszystkich metod* należących do tego *interfejsu*.

Niektóre interfejsy mają więcej niż jedną metodę, gdyż istnieje więcej rodzajów zdarzeń. Na przykład, implementując interfejs `MouseListener`, uzyskujemy możliwość obsługi zdarzeń `mousePressed`, `mouseReleased`, `mouseMoved` i tak dalej. Każdemu z tych zdarzeń odpowiada jedna metoda interfejsu i to niezależnie od tego, że każda z nich przyjmuje argument typu `MouseEvent`. Jeśli zaimplementujesz interfejs `MouseListener`, to metoda `mousePressed()` zostanie wywołana, gdy (tak, zgadłeś) użytkownik naciśnie przycisk myszy. A gdy użytkownik zwolni przycisk, zostanie wywołana metoda `mouseReleased()`. A zatem, w przypadku zdarzeń związanych z myszą istnieje tylko jeden *obiekt* zdarzenia, lecz kilka metod reprezentujących różne *rodzaje* zdarzeń.

Implementując interfejs odbiorcy dajesz przyciskowi możliwość wywołania Twojego kodu. To właśnie do tego interfejsu należy wywoływana metoda zwrotna.



W jaki sposób komunikuję się źródło zdarzeń oraz ich odbiorca?



Odbiorca

Jeśli chcesz dowiadywać się o zdarzeniach ActionEvent generowanych przez przycisk, powinieneś zaimplementować interfejs ActionListener. Przycisk musi wiedzieć, że interesujesz się tymi zdarzeniami i dlatego powinieneś się w nim zarejestrować. W tym celu wywołaj metodę addActionListener(this) tego przycisku i przekaż do niej odwołanie typu ActionListener (w tym przypadku to Twój obiekt jest typu ActionListener, zatem w wywoaniu metody addActionListener() możesz przekazać odwołanie *this*). Przycisk musi dysponować jakimś sposobem wywołania Twojego kodu w momencie zajścia zdarzenia, zatem wywołuje odpowiednią metodę interfejsu odbiorcy. Jako ActionListener Twoja klasa *musi* implementować jedyną metodę tego interfejsu — actionPerformed(). O spełnienie tego wymogu zadba kompilator.

Źródło zdarzeń

Przycisk jest źródłem zdarzeń ActionEvent, zatem musi wiedzieć, jakie obiekty są zainteresowane ich odbieraniem. Przycisk ma metodę addActionListener(), której zainteresowane obiekty mogą użyć, aby poinformować przycisk, że chcą odbierać jego zdarzenia.

W przypadku wywołania metody `addActionListener()` (przez potencjalnego odbiorcę zdarzeń), przycisk pobiera jej parametr (czyli odwołanie do obiektu odbiorcy) i zapisuje jego wartość na liście. Kiedy użytkownik kliknie przycisk, ten „zgłasza” zdarzenie, wywołując metody `actionPerformed()` wszystkich odbiorców zarejestrowanych na liście.

Przechwytywanie zdarzeń

Przechwytywanie zdarzenia

ActionEvent przycisku:

- 1 Zaimplementuj interfejs ActionListener.
- 2 Zarejestruj odbiorcę zdarzeń (poinformuj przycisk, że chcesz odbierać zdarzenia).
- 3 Zdefiniuj metodę obsługującą zdarzenia (czyli metodę actionPerformed() interfejsu ActionListener).

```
import javax.swing.*;           Nowa instrukcja importu określająca
import java.awt.event.*;         pakiet, do którego należy interfejs
                                ActionListener oraz klasa ActionEvent.  
  
1                                         ↓  
public class ProstyGUI1b implements ActionListener {  
    JButton przycisk;  
  
    public static void main(String[] args) {  
        ProstyGUI1b apGUI = new ProstyGUI1b();  
        apGUI.doDziela();  
    }  
  
    public void doDziela() {  
        JFrame ramka = new JFrame();  
        przycisk = new JButton("Kliknij mnie");  
  
        2 - przycisk.addActionListener(this);  
  
        ramka.getContentPane().add(przycisk);  
        ramka.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        ramka.setSize(300,300);  
        ramka.setVisible(true);  
    }  
  
    3  
    public void actionPerformed(ActionEvent zdarzenie) {  
        przycisk.setText("Zostałem klinięty");  
    }  
}
```

Zaimplementuj ten interfejs. Informuje on, że: „obiekt klasy ProstyGUI1b JEST typu ActionListener”.
(Przycisk będzie przekazywać zdarzenia wyłącznie do klas implementujących interfejs ActionListener).

Zarejestruj się w przycisku. W ten sposób przekażesz przyciskowi informację: „Dodaj mnie na listę odbiorców zdarzeń”. Przekazywanym argumentem MUSI być obiekt klasy, która implementuje interfejs ActionListener!!!

Zaimplementuj metodę actionPerformed() interfejsu ActionListener. To właśnie ona służy do obsługi zdarzeń.

Przycisk wywołuje tę metodę, aby poinformować Cię, że zaszło zdarzenie. Do metody przekazywany jest obiekt ActionEvent, jednak w tym przykładzie nie potrzebujemy go. Wystarczy, że dowiemy się o samym fakcie zajścia zdarzenia.

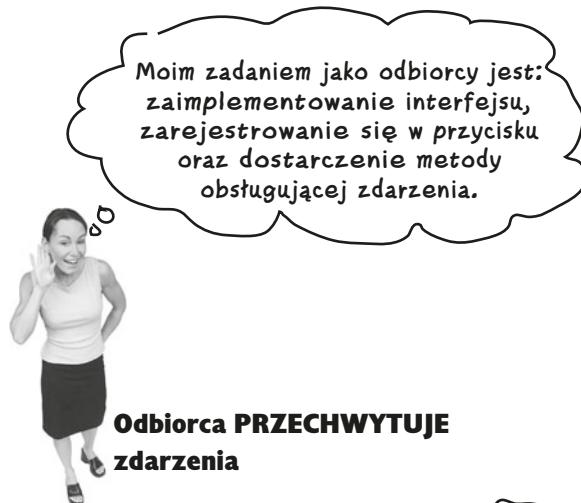
Odbiorcy, źródła i zdarzenia

W przeważającej większości przypadków błyskotliwy rozwój Twojej kariery nie będzie wymagał, abyś stawał się źródłem zdarzeń.

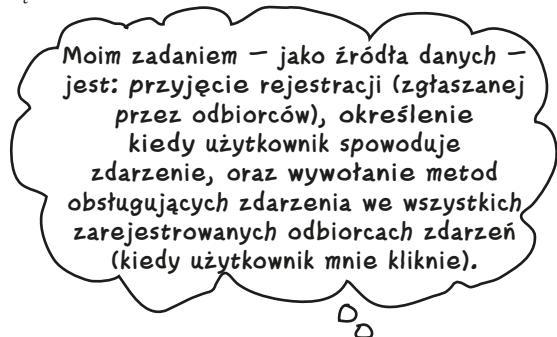
(Niezależnie od tego, jak mocno wyobrażasz sobie siebie samego w centrum swojego towarzyskiego wszechświata).

Pogódź się z tym. *Twoim zadaniem jest pełnienie roli dobrego odbiorcy.*

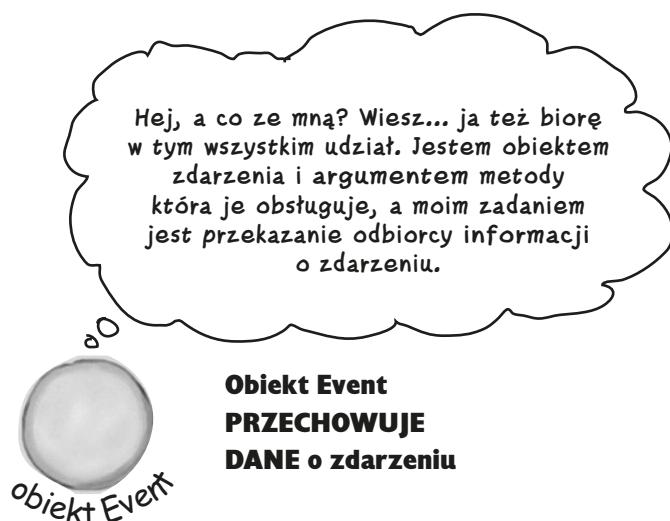
(Co w znacznej mierze może poprawić Twój życie towarzyskie, jeśli będziesz to robić naprawdę szczerze).



Odbiorca PRZECHWYTUJE zdarzenia



Źródło WYSYŁA zdarzenia



**Obiekt Event
PRZECHOWUJE
DANE o zdarzeniu**

Obsługa zdarzeń

Nie istnieja
głupie pytania

P: Dlaczego nie mogę być źródłem zdarzeń?

O: Ależ MOŻESZ. Napisaliśmy jedynie, że w większości przypadków będziesz odbiorcą zdarzeń, a nie ich twórcą i źródłem (przynajmniej w początkowym okresie Twojej błyskotliwej kariery jako programisty). Większość zdarzeń, które mogą Cię interesować, jest zgłaszana przez klasy tworzące Java API, a całe Twoje zadanie ogranicza się do ich odbierania i obsługi. Możesz jednak zaprojektować program, w którym będziesz potrzebować własnych zdarzeń; na przykład, zdarzenia KursAkci j Event, które będzie zgłaszane, gdy aplikacja analizująca kursy akcji odkryje coś, co uznasz za interesujące. W takim przypadku stworzyłbyś zapewne obiekt Anal i zatorKursuAkci j i pełniący funkcję źródła zdarzeń i zrobił to, co robi przycisk (oraz każde inne źródło) — stworzył interfejs odbiorcy wykorzystujący Twoje zdarzenie, udostępnił metodę rejestracji odbiorców i dodawał odbiorców do listy w momencie wywołania tej metody. Następnie, kiedy zajdzie zdarzenie, tworzyłbyś obiekt KursAkci j Event (to jeszcze jedna klasa, którą musiałbyś napisać) i przekazywał go do odbiorców zarejestrowanych na liście, wywołując w tym celu ich metodę kursAkci j Zmieni ony (KursAkci j Event kae). I nie zapomnij, że dla każdego rodzaju zdarzenia musisz stworzyć odpowiadający mu interfejs odbiorcy (zatem stworzyłbyś zapewne interfejs KursAkci j Listener zawierający metodę kursAkci j Zmieni ony())



Zaostrz ołówek

Każdy z wymienionych poniżej komponentów interfejsu użytkownika jest źródłem jednego lub kilku zdarzeń. Dopasuj komponenty oraz zdarzenia, które mogą być przez nie generowane. Niektóre komponenty mogą być źródłem więcej niż jednego zdarzenia, a niektóre zdarzenia mogą być generowane przez więcej niż jeden komponent.

Komponenty

- pole wyboru
- pole tekstowe
- lista przewijana
- przycisk
- okno dialogowe
- przycisk opcji
- opcja menu

Metody obsługi zdarzeń

- windowClosing()
- actionPerformed()
- itemStateChanged()
- mousePressed()
- keyTyped()
- mouseExited()
- focusGained()

P: Nie rozumiem, jakie znaczenie ma obiekt zdarzenia przekazywany do metody obsługującej zdarzenia. Gdyby ktoś wywołał moją metodę mousePressed, to jakich innych informacji bym potrzebował.

O: W wielu przypadkach większość projektantów interfejsu użytkownika nie potrzebuje obiektu zdarzenia. To nic innego jak niewielki nośnik danych, którego celem jest przekazanie większej ilości informacji dotyczących zdarzenia. Jednak czasami możesz pobrać z tego obiektu szczegółowe informacje o zdarzeniu. Na przykład w razie wywołania metody mousePressed(), będziesz wiedzieć, że naciśnięty został jeden z przycisków myszy. Ale co zrobić, jeśli będziesz chciał się dowiedzieć, gdzie w tym momencie znajdował się wskaźnik myszy? Innymi słowy, jak możesz określić współrzędne x i y miejsca, w którym znajdował się wskaźnik myszy w momencie naciśnięcia jej przycisku?

Może się także zdarzyć, że będziesz chciał zarejestrować tego samego odbiorcę w kilku źródłach zdarzeń. Na przykład program kalkulatora ma dziesięć przycisków z cyframi, a ponieważ wszystkie te przyciski działają dokładnie tak samo, więc zapewne nie będziesz chciał tworzyć odrębnego odbiorcy dla każdego z nich. Zamiast tego mógłbyś zarejestrować tego samego odbiorcę w każdym z dziesięciu przycisków, a następnie — po odebraniu zdarzenia (czyli wywołaniu metody obsługującej) — mógłbyś użyć obiektu zdarzenia do wywołania jakiejś metody i określenia, jakie było prawdziwe źródło zdarzenia. Innymi słowy, mógłbyś go użyć do określenia, który przycisk wygenerował zdarzenie.

Skąd można się DOWIEDZIEĆ, że obiekt jest źródłem zdarzeń?

Zajrzyj do dokumentacji API.

W porządku... ale czego mam szukać?

Metody, której nazwa zaczyna się od liter „add”, a kończy „Listener”, i której argument jest interfejsem odbiorcy. Jeśli zatem zauważysz metodę:

`addKeyListener(KeyListener k)`

będziesz wiedzieć, że klasa posiadająca tę metodę jest źródłem zdarzeń KeyEvent. Zawsze obowiązuje właśnie taki system nazewnictwa.

Wróćmy z powrotem do graficznego interfejsu użytkownika

Dowiedzieliśmy się już nieco o tym, jak działają zdarzenia (niedługo dowiemy się więcej na ten temat), wróćmy zatem do wyświetlania na ekranie różnych rzeczy. Zanim wrócimy do obsługi zdarzeń, poświęcimy parę minut na przedstawienie zabawnych sposobów wyświetlania grafiki.

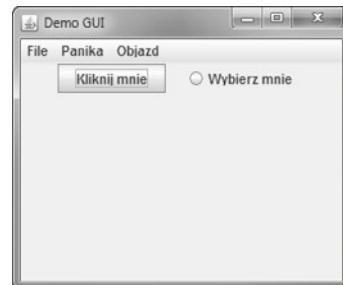
Trzy sposoby na wyświetlanie różnego typu elementów w programie o graficznym interfejsie użytkownika:

1 Umieszczenie komponentu w ramce.

W ten sposób możesz wyświetlać przyciski, listy, przyciski opcji i tak dalej.

```
ramka.getContentPane().add(mojPrzycisk);
```

Pakiet javax.swing udostępnia co najmniej kilkanaście różnych komponentów.

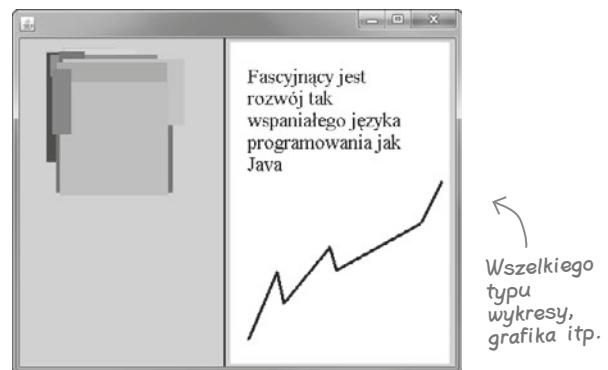


2 Narysowanie na komponentie dwuwymiarowej grafiki.

Möesz użyć obiektu graphics, aby rysować różne figury.
`graphics.fillOval(70,70,100,100);`

Möażna rysować znacznie więcej figur niż tylko okręgi i prostokąty, w Java2D API znajdziesz wiele interesujących i wyrafinowanych metod do tworzenia grafiki.

Grafika, gry,
symulacje itp.



3 Wyświetlenie w komponencie obrazu w formacie JPEG.

Na komponentach można także wyświetlać swoje własne obrazy.

```
graphics.drawImage(mojeZdj,10,10,this);
```



Stwórz własny komponent umożliwiający rysowanie

Jeśli chcesz wyświetlić na ekranie własną grafikę, najlepszym rozwiązaniem będzie stworzenie własnego elementu interfejsu użytkownika pozwalającego na rysowanie. Taki element będzie można umieścić w ramce tak samo jak przycisk lub jakkolwiek inny komponent, lecz kiedy zostanie wyświetlony, będzie prezentować określony obrazek. Obrazki prezentowane na takim elemencie mogą się nawet przesuwać, tworząc animację, możesz także zmieniać kolor tła za każdym razem, gdy użytkownik kliknie przycisk.

To banalnie proste zadanie.

Stwórz klasę potomną klasy JPanel i przesłonij jedną, jedyną metodę — paintComponent().

Cały kod odpowiedzialny za stworzenie i prezentację grafiki trzeba umieścić w metodzie `paintComponent()`. Możesz sobie wyobrazić, że system wywołuje tę metodę, aby powiedzieć: „Hej, komponencie, czas narysować swoją zawartość”. Jeśli chcesz narysować okrąg, to metoda `paintComponent()` będzie zawierać kod rysujący okrąg. Kiedy ramka zawierająca Twój panel zostanie wyświetlona, jednocześnie zostanie wywołana metoda `paintComponent()`, która narysuje okrąg. Kiedy użytkownik zminimalizuje okno, wirtualna maszyna Javy domyśli się, że po przywróceniu wcześniejszych wymiarów ramki konieczne będzie powtórne wyświetlenie jej zawartości, zatem ponownie wywoła metodę `paintComponent()`. Za każdym razem, gdy wirtualna maszyna Javy uzna, że konieczne jest odświeżenie zawartości komponentu, wywoła jego metodę `paintComponent()`.

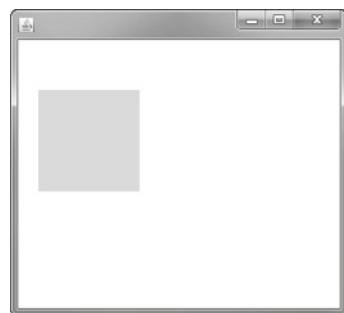
I jeszcze jedna sprawa, **metody `paintComponent()` nigdy nie powinieneś wywoływać sam!** Argument przekazywany w wywoaniu tej metody (obiekt `Graphics`) jest tak zwanym „płotnem”, którego zawartość zostaje przekazana i wyświetlona na prawdziwym ekranie monitora. Tego obiektu nie możesz sam zdobyć — jedynie system może Ci go przekazać. W dalszej części rozdziału przekonasz się jednak, że możesz poprosić system o odświeżenie komponentu (przy użyciu metody `repaint()`), co w rezultacie powoduje wywołanie metody `paintComponent()`.

```
import java.awt.*; ← Potrzebujesz obu tych pakietów.  
import javax.swing.*; ←  
  
class MojPanelGraficzny extends JPanel {  
  
    public void paintComponent(Graphics g) {  
        g.setColor(Color.orange); ←  
        g.fillRect(20,50,100,100); ←  
    }  
}
```

Stwórz klasę potomną klasy JPanel — komponent, który będziesz mógł dodać do ramki, tak samo jak cokolwiek innego, z tą różnicą, że będzie to Twój własny komponent, dostosowany do Twoich potrzeb.

To jest bardzo ważna metoda graficzna. Nigdy NIE BĘDZIESZ jej wywoływać sam. To system ją wywołuje i mówi: „Oto piękna, nowa powierzchnia do rysowania typu Graphics, teraz możesz sobie na niej rysować”.

Wyobraź sobie, że „g” jest tajemniczą maszyną do rysowania. Najpierw określasz, jakiego koloru ma używać podczas rysowania, a następnie, co ma narysować (współrzędne początku oraz współrzędne figury).



Fajne operacje, jakie można realizować w metodzie paintComponent()

Przyjrzyjmy się kilku innym operacjom, jakie można wykonywać w metodzie `paintComponent()`. Najciekawszą zabawę zagwarantują Ci dopiero eksperymenty wykonywane samodzielnie. Spróbuj zmieniać liczby i przejrzyj dokumentację API dla klasy `Graphics` (w dalszej części książki przekonasz się, że nasze możliwości wykraczają poza to, na co pozwala klasa `Graphics`).

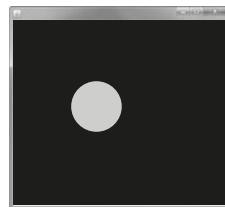
Wyświetlanie plików JPEG

```
public void paintComponent(Graphics g) { ← Tutaj podajesz nazwę pliku.
    Image obrazek = new ImageIcon("kotzilla.jpg").getImage(); ←
    g.drawImage(obrazek, 3, 4, this); ← Współrzędne x i y określające położenie lewego
}                                     ↑   górnego wierzchołka wyświetlanego obrazka. W tym
                                         ↑   przypadku oznaczają one: „3 piksele na prawo
                                         ↑   od lewej krawędzi panelu oraz 4 piksele poniżej
                                         ↑   jego górnej krawędzi”. Te liczby zawsze określają
                                         ↑   położenie względem komponentu (w naszym
                                         ↑   przypadku — względem obiektu klasy rozszerzającej
                                         ↑   JPanel), a nie całej ramki.
```



Rysowanie koła wypełnionego losowo wybranym kolorem i widocznego na czarnym tle.

```
public void paintComponent(Graphics g) { ← Wypełnij cały panel czarnym tłem
}                                     ↑   (czarny jest kolorem domyślnym).
```



```
g.fillRect(0,0,this.getWidth(),this.getHeight());
```

Pierwsze dwa argumenty definiują współrzędne (x,y) lewego górnego wierzchołka obszaru wyrażone względem panelu; zatem 0,0 oznacza: „zaczni 0 pikseli na prawo od lewej krawędzi panelu i 0 pikseli poniżej jego górnej krawędzi”. Pozostałe dwa argumenty informują: „szerokość obszaru ma być taka sama jak szerokość panelu (`this.getWidth()`), podobnie jak wysokość obszaru, która ma odpowiadać wysokości panelu (`this.getHeight()`)”.

```
int czerwony = (int) (Math.random() * 256);
int zielony = (int) (Math.random() * 256);
int niebieski = (int) (Math.random() * 256);

Color kolorLosowy = new Color(czerwony, zielony, niebieski);
g.setColor(kolorLosowy);
g.fillOval(70,70,100,100);
```

↑
Zacznij rysować 70 pikseli na prawo od lewej krawędzi i 70 pikseli poniżej górnej krawędzi, a oval ma mieć 100 pikseli szerokości i tyle samo wysokości.

Kolor można utworzyć, przekazując trzy liczby całkowite określające wartości RGB (czerwonego, zielonego i niebieskiego).

Odwołanie do porządkowej klasy Graphics ukrywa obiekt Graphics2D

Argument metody `paintComponent()` został zadeklarowany jako obiekt klasy `Graphics` (`java.awt.Graphics`).

```
public void paintComponent(Graphics g) { }
```

A zatem, parametr „`g`” JEST obiektem `Graphics`. Co oznacza, że mógłby on także być obiektem klasy *potomnej* `Graphics` (dzięki możliwościom, jakie zapewnia polimorfizm). I okazuje się, że w rzeczywistości tak właśnie jest.

W rzeczywistości obiekt zapisywany w parametrze „`g`” jest obiektem klasy `Graphics2D`.

Czemu to takie ważne? Ponieważ odwołanie `Graphics2D` daje nam większe możliwości niż odwołanie `Graphics`. Obiekt klasy `Graphics2D` ma większe możliwości niż obiekt klasy `Graphics`, a w rzeczywistości za odwołaniem `Graphics` kryje się obiekt klasy `Graphics2D`.

Pamiętaj o polimorfizmie. Kompilator określa, jakie metody można wywoływać na podstawie typu odwołania, a nie typu obiektu. Jeśli dysponujesz obiektem klasy `Pies` zapisanym w zmiennej referencyjnej typu `Zwierze`:

```
Zwierze z = new Pies();
```

nie możesz użyć poniższego wywołania:

```
z.szczerkaj();
```

Pomimo to, że wiesz, że w zmiennej jest zapisany obiekt `Pies`. Kompilator przyjrzy się zmiennej, zauważ, iż jest ona typu `Zwierze`, po czym sprawdzi, że na pilocie klasy `Zwierze` nie ma żadnego przycisku odpowiadającego metodzie `szczerkaj()`. Niemniej jednak wciąż możemy odzyskać obiekt `Pies`, którym on przecież faktycznie jest:

```
Pies p = (Pies) z;
p.szczerkaj();
```

A zatem, sprawy dotyczące obiektu `Graphics` mają się następująco:

Jeśli chcesz używać metod klasy `Graphics2D`, nie możesz posługiwać się bezpośrednio parametrem „`g`” metody `paintComponent()`. Jednak możesz go rzutować i zapisać w nowej zmiennej typu `Graphics2D`.

```
Graphics2D g2d = (Graphics2D) g;
```

Metody, których możesz używać, posługując się odwołaniem klasy `Graphics`:

```
drawImage()
drawLine()
drawPolygon()
drawRect()
drawOval()
fillRect()
fillRoundRect()
setColor()
```

Aby rzutować obiekt `Graphics2D` i zapisać go w odwołaniu tego samego typu:

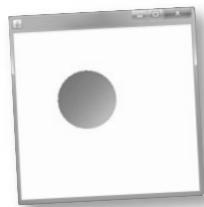
```
Graphics2D g2d = (Graphics2D) g;
```

Metody, których możesz używać, posługując się odwołaniem klasy `Graphics2D`:

```
fill3DRect()
draw3DRect()
rotate()
scale()
shear()
transform()
setRenderingHints()
```

(To nie są wszystkie metody udostępniane przez te klasy, kompletne listy możesz znaleźć w dokumentacji API)

Życie jest zbyt krótkie, by rysować koło o jednolitym kolorze, jeśli można użyć gradientu



```
public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;
}
```

W rzeczywistości jest to obiekt Graphics2D skromnie udający obiekt Graphics.

Wykonujemy rzutowanie, abyśmy mogli skorzystać z możliwości, które są dostępne wyłącznie w obiekcie Graphics2D

```
GradientPaint gradient = new GradientPaint(70,70, Color.blue, 150,150, Color.orange);
```

Punkt początkowy. Kolor początkowy. Punkt końcowy. Kolor końcowy.

```
g2d.setPaint(gradient);
g2d.fillOval(70,70,100,100);
}
```

To wywołanie sprawia, że wirtualny pędzel będzie używać gradientu, a nie jednolitego koloru.

Wywołanie metody fillOval() tak naprawdę oznacza: „wypełnij oval wzorem, jaki aktualnie jest używany przez pędzel (to znaczy gradientem)”.

```
public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;

    int czerwony = (int) (Math.random() * 256);
    int zielony = (int) (Math.random() * 256);
    int niebieski = (int) (Math.random() * 256);
    Color kolorPoczatku = new Color(czerwony, zielony, niebieski);

    czerwony = (int) (Math.random() * 256);
    zielony = (int) (Math.random() * 256);
    niebieski = (int) (Math.random() * 256);
    Color kolorKonca = new Color(czerwony, zielony, niebieski);

    GradientPaint gradient = new GradientPaint(70,70,kolorPoczatku, 150,150, kolorKonca);

    g2d.setPaint(gradient);
    g2d.fillOval(70,70,100,100);
}
```

Ta metoda paintComponent() przypomina poprzednią z tym, że kolor początkowy i końcowy gradientu są wybierane losowo. Sprawdź i sam się przekonaj!

**ZDARZENIA**

- Aby stworzyć graficzny interfejs użytkownika, zacznij od stworzenia okna; zazwyczaj będzie to obiekt JFrame:
JFrame ramka = new JFrame();
- W poniższy sposób możesz dodawać od obiektu JFrame komponenty (takie jak przyciski, pola tekstowe itd.):
ramka.getContentPane().add(przycisk);
- Obiekt JFrame, podobnie jak większość innych komponentów, nie pozwala na bezpośrednie dodawanie do niego innych elementów interfejsu użytkownika — w tym celu należy się posłużyć panelem zawartości.
- Aby wyświetlić okno (JFrame), należy podać jego wymiary i określić, że jest widoczne:
ramka.setSize(300,300);
ramka.setVisible(true);
- Aby się dowiedzieć, kiedy użytkownik kliknie przycisk (lub wykona jakąkolwiek inną operację na elementach interfejsu użytkownika), musisz odbierać zdarzenia.
- Aby odbierać zdarzenia, musisz poinformować źródło zdarzeń, że jesteś nimi zainteresowany. Źródło zdarzeń to komponent (przycisk, pole wyboru itd.), który generuje zdarzenia na podstawie czynności wykonywanych przez użytkownika.
- Interfejs odbiorcy pozwala komponentowi będącemu źródłem zdarzeń na wywołanie Twojego kodu; jest to możliwe dzięki temu, iż interfejs definiuje metodę (lub metody), które źródło zdarzeń będzie wywoływać w momencie zajścia zdarzenia.
- Aby się zarejestrować w źródle zdarzeń, należy wywołać odpowiednią metodę rejestracyjną. Nazwy tych metod zawsze są tworzone według wzoru: add<typZdarzenia>Listener. Na przykład, aby się zarejestrować i odbierać zdarzenia ActionEvent generowane przez przycisk, należy wywołać metodę: **przycisk.addActionListener(this);**
- Zaimplementuj interfejs odbiorcy zdarzeń, tworząc wszystkie jego metody. Kod obsługujący zdarzenia umieść w odpowiedniej metodzie tego interfejsu. Poniżej przedstawiliśmy przykład metody obsługującej zdarzenia ActionEvent:

```
public void actionPerformed(ActionEvent zdarzenie) {
    przycisk.setText("kliknąłeś!!!");
}
```

- Obiekt przekazywany do metody obsługującej zdarzenia zawiera informacje dotyczące zdarzenia, w tym jego źródło.

GRAFIKA

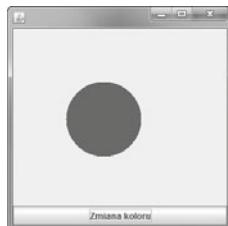
- Dwuwymiarową grafikę możesz rysować bezpośrednio na komponentach.
- Zawartość plików .gif oraz .jpg możesz wyświetlać bezpośrednio na komponentach.
- Aby wyświetlać własną grafikę (w tym także zawartość plików .gif oraz .jpg), stwórz klasę potomną klasy JPanel i prześroń w niej metodę paintComponent().
- Metoda paintComponent() jest wywoływana bezpośrednio przez system zarządzający graficznym interfejsem użytkownika. TEJ METODY NIGDY NIE WYWOLUJESZ SAM! Argumentem metody paintComponent() jest obiekt Graphics zapewniający dostęp do powierzchni, na której można rysować. Zawartość tej powierzchni trafi bezpośrednio na ekran. Obiekt Graphics nie można stworzyć samemu.
- Oto przykłady typowych metod obiektu Graphics (parametru metody paintComponent()):
g.setColor(Color.blue);
g.fillRect(20,40,100,100);
- Aby wyświetlić obrazek zapisany w pliku .jpg, należy stworzyć obiekt Image:
Image obrazek = new ImageIcon("kotzilla.jpg").getImage();
a następnie "narysować" jego zawartość:
g.drawImage(obrazek,3,4,this);
- Obiekt, do którego odwołuje się parametr Graphics metody paintComponent(), jest w rzeczywistości obiektem klasy Graphics2D. Klasa ta udostępnia wiele metod, oto niektóre z nich: fill3DRect(), draw3DRect(), rotate(), scale(), shear() oraz transform().
- Aby móc skorzystać z metod klasy Graphics2D, należy rzutować parametr metody paintComponent() na ten typ:
Graphics2D g2d = (Graphics2D) g;

Możemy przechwytywać zdarzenia.

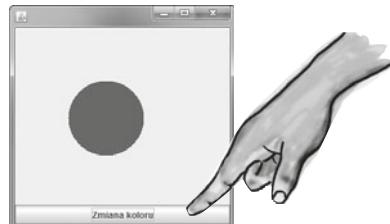
Możemy rysować.

Ale czy możemy coś narysować w momencie przechwycenia zdarzenia?

Wykorzystajmy zdarzenie, aby zmieniać rysunek wyświetlany na panelu. Chcemy, aby za każdym razem, gdy użytkownik kliknie przycisk, zmieniał się kolor kółka prezentowanego na panelu. Oto jak działa nasz program.



- 1 Tworzona jest ramka zawierająca dwa komponenty (nasz rysunkowy panel oraz przycisk). Tworzony jest odbiorca zdarzeń, który następnie zostaje zarejestrowany w przycisku. Ramka jest wyświetlana i czeka, aż użytkownik kliknie przycisk.



- 2 Użytkownik kliką przycisk, a ten tworzy obiekt zdarzenia i wywołuje procedurę obsługi odbiorcy zdarzeń.



- 3 Odbiorca zdarzeń wywołuje metodę repaint() ramki. System zarządzający interfejsem użytkownika wywołuje metodę paintComponent() naszego panelu rysunkowego.

- 4 I proszę! Kolor prezentowanego kółka zmienia się, gdyż ponownie została wywołana metoda paintComponent(), która za każdym razem losowo określa jego kolor.



Układy GUI — wyświetlanie w ramce więcej niż jednego komponentu

Układy graficznego interfejsu użytkownika (w skrócie układy GUI) zostaną szczegółowo opisane w *następnym* rozdziale. Jak na razie przedstawimy jedynie krótki opis tego zagadnienia, abyś mógł kontynuować. Ramka domyślnie ma pięć regionów, w których możesz umieszczać komponenty. W każdym z nich możesz umieścić tylko *jeden* komponent. Ale bez paniki! Jednym z tych komponentów może być panel zawierający trzy kolejne komponenty, z których jednym może być kolejny panel zawierający dwa komponenty, z których jednym... już pewnie rozumiesz zasadę. W rzeczywistości trochę „oszukujemy”, dodając komponenty do ramki w poniższy sposób:

```
ramka.getContentPane().add(przycisk);
```

W rzeczywistości nie powinieneś tego robić w taki sposób (wywoływać metody pobierającej jeden argument).

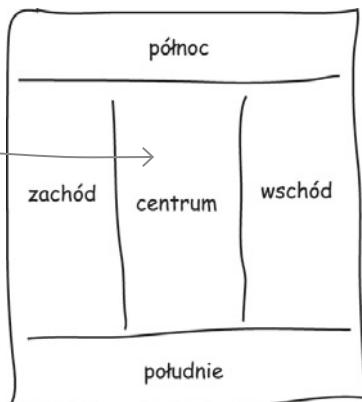
```
ramka.getContentPane().add(BorderLayout.CENTER, przycisk);
```

Wywołujemy dwuargumentową wersję metody add(), która pobiera stałą określającą region oraz komponent, który należy w tym regionie umieścić.

Oto znacznie lepsza (i zazwyczaj obowiązkowa) metoda dodawania komponentów do domyślnego panelu zawartości ramki. Zawsze należy określić, GDZIE (czyli w którym regionie) komponent ma zostać umieszczony.

W przypadku wywołania metody add() pobierającej jeden argument (czyli tej, której nie powinniśmy używać) komponent automatycznie zostanie umieszczony w regionie centralnym.

region domyślny



Zaostrz ołówek

Na podstawie rysunków przedstawionych na stronie 395 napisz kod, który umieści w ramce panel i przycisk.



Kółko zmienia kolor za każdym razem, gdy klikasz przycisk.

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ProstyGUI1c implements ActionListener {
    JFrame ramka;

    public static void main(String[] args) {
        ProstyGUI1c apGUI = new ProstyGUI1c();
        apGUI.doDziela();
    }

    public void doDziela() {
        ramka = new JFrame();
        ramka.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton przycisk = new JButton("Zmiana koloru");
        przycisk.addActionListener(this);
    }

    MojPanelRysunkowy panelR = new MojPanelRysunkowy();

    ramka.getContentPane().add(BorderLayout.SOUTH, przycisk);
    ramka.getContentPane().add(BorderLayout.CENTER, panelR);
    ramka.setSize(300,300);
    ramka.setVisible(true);
}

public void actionPerformed(ActionEvent zdarzenie) {
    ramka.repaint();
}

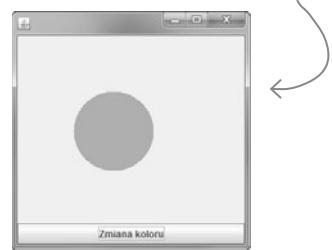
```

```

class MojPanelRysunkowy extends JPanel {
    public void paintComponent(Graphics g) {
        // kod wyświetlający koło wypełnione losowym kolorem
        // został przedstawiony we wcześniejsze części rozdziału
        // na stronie 391
    }
}

```

Niestandardowy panel rysunkowy (obiekt *MojPanelRysunkowy*) znajduje się w CENTRALNYM regionie ramki.



Przycisk znajduje się w POŁUDNIOWYM regionie ramki.

Dodajemy odbiorcę zdarzeń do przycisku (jest nim bieżący obiekt — this).

Do ramki dodajemy dwa komponenty (przycisk oraz panel rysunkowy), w dwóch różnych regionach.

Gdy użytkownik kliknie, wywoływana jest metoda *repaint()*, która nakazuje ponowne wyświetlenie zawartości ramki. Oznacza to wywołanie metody *paintComponent()* każdego komponentu umieszczonego w ramce.

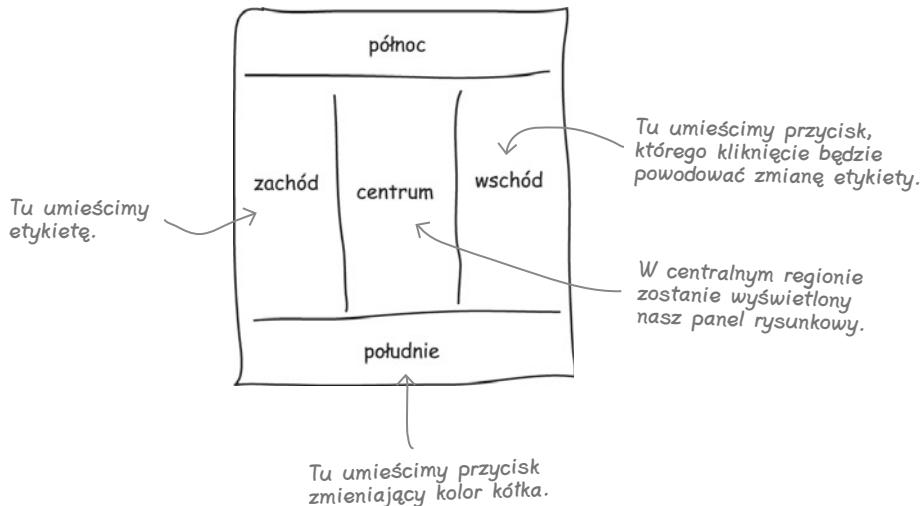
Metoda *paintComponent()* panelu jest wywoływana za każdym razem, gdy użytkownik kliknie przycisk.

Stosowanie wielu odbiorców zdarzeń

Spróbujmy użyć DWÓCH przycisków

Południowy region będzie działać tak samo jak dotychczas, czyli będzie powodować ponowne wyświetlenie zawartości całej ramki. Drugi przycisk (który umieścimy we wschodnim regionie) będzie powodował zmianę tekstu etykiety (etykieta to zwyczajny tekst wyświetlany na ekranie).

Zatem potrzebujemy CZTERECH komponentów



Teraz będziemy potrzebować DWÓCH zdarzeń

O rany!

Czy to w ogóle jest możliwe? W jaki sposób można obsługiwać dwa zdarzenia, skoro można mieć tylko jedną metodę `actionPerformed()`?

Ten przycisk zmienia tekst wyświetlany po drugiej stronie okna.

Jak można przechwytywać zdarzenia generowane przez dwa różne przyciski, z których każdy musi wykonywać różne operacje?

1 Rozwiążanie pierwsze:

Zaimplementuj dwie metody actionPerformed()

```
class MojGUI implements ActionListener {
    // dużo innego kodu a następnie:
    public void actionPerformed(ActionEvent zdarzenie) {
        ramka.repaint();
    }

    public void actionPerformed(ActionEvent zdarzenie) {
        etykieta.setText("To boli!");
    }
}
```

Wada: Tego nie można zrobić! W klasie Javy nie można dwukrotnie zaimplementować tej samej metody. Takiej klasy nie da się skompilować. A nawet gdyby coś takiego *dało się* zrobić, to skąd źródło danych wiedziałoby, którą metodę należy wywołać?

2 Rozwiążanie drugie:

Zarejestruj tego samego odbiorcę w dwóch przyciskach.

```
class MojGUI implements ActionListener {
    // deklaracje kilku składowych
    public void doRoboty() {
        // tworzenie graficznego interfejsu użytkownika
        przyciskKolor = new JButton();
        przyciskEtykieta = new JButton();
        przyciskKolor.addActionListener(this); ← Zarejestruj tego samego
        przyciskEtykieta.addActionListener(this); ← odbiorcę w dwóch
                                                przyciskach.
        // dalsza część kodu ...
    }

    public void actionPerformed(ActionEvent zdarzenie) {
        if (zdarzenie.getSource() == przyciskKolor) {
            ramka.repaint(); ← Proszę obiekt zdarzenia, aby
        } else {                                sprawdzić, który przycisk go
            etykieta.setText("To boli!");       zgłosił, i na tej podstawie
        }
    }
}
```

Wada: To rozwiązanie działa, jednak w większości przypadków nie jest przykładem dobrego programowania obiektowego. W tym przypadku jedna metoda obsługi zdarzeń musi wykonywać dwie różne czynności. Konieczność zmiany sposobu obsługi jednego źródła danych oznacza, że będziesz musiał wprowadzić modyfikacje w kodzie metody obsługi zdarzeń wywoływanej przez wszystkie źródła. Czasami takie rozwiązanie jest dobre, jednak zazwyczaj utrudnia ono utrzymanie kodu oraz jego rozbudowę.

Stosowanie wielu odbiorców zdarzeń

Jak można przechytywać zdarzenia generowane przez dwa różne przyciski, z których każdy musi wykonywać różne operacje?

3 Rozwiążanie trzecie:

Utwórz dwie niezależne klasy implementujące interfejs ActionListener.

```
class MojGUI {  
    JFrame ramka;  
    JLabel etykieta;  
    void gui() {  
        // kod tworzący obiekty dwóch odbiorców zdarzeń i rejestrujący  
        // jednego z nich w jednym przycisku, a drugiego w drugim  
    }  
} // koniec klasy
```

```
class OdbiorcaPrzyciskuKoloru implements ActionListener {  
    public void actionPerformed(ActionEvent zdarzenie) {  
        ramka.repaint();  
    }  
}
```

To nie zadziała! Ta klasa nie dysponuje odwołaniem do składowej „ramka” klasy MojGUI.

```
class OdbiorcaPrzyciskuEtykiety implements ActionListener {  
    public void actionPerformed(ActionEvent zdarzenie) {  
        etykieta.setText("To boli!");  
    }  
}
```

Mamy problem! Ta klasa nie ma odwołania do składowej „etykieta”.

Wada: Te klasy nie mają dostępu do składowych, na których muszą operować, czyli „ramka” oraz „etykieta”.

Problem ten można rozwiązać, lecz wymagałoby to przekazania do każdej z klas odbiorów zdarzeń odwołania do głównej klasy zawierającej elementy interfejsu użytkownika. Dzięki temu odwołaniu klasa odbiorcy zdarzeń mogłaby w metodzie actionPerformed() uzyskać dostęp do składowych klasy definiującej interfejs użytkownika. Jednak takie rozwiązanie stoi w sprzeczności z zasadami hermetyzacji, zatem prawdopodobnie musiałby stworzyć odpowiednie metody zwracające dla poszczególnych elementów interfejsu (na przykład getRamka(), getEtykieta() i tak dalej). Prawdopodobnie musiałby także stworzyć konstruktor dla klasy odbiorcy zdarzeń, aby przekazywać do niej odwołanie do klasy zawierającej elementy interfejsu użytkownika. Jak widać jest to trochę zagmatwane i skomplikowane.

Musi istnieć jakieś lepsze rozwiązanie!



Czyż nie byłoby cudownie, gdyby można było stworzyć dwie różne klasy odbiorców zdarzeń, lecz gdyby klasy te miały dostęp do składowych głównej klasy definiującej interfejs użytkownika? To prawie tak, jak gdyby klasy odbiorców należały do innej klasy. W takiej sytuacji mielibyśmy wszystko, co nam potrzebne. O tak, to byłoby wspaniałe. Ale to pewnie tylko fantazja...

Klasy wewnętrzne spieszają z pomocą!

Można stworzyć jedną klasę *wewnątrz* innej. To całkiem łatwe. Po prostu upewnij się, że definicja klasy wewnętrznej znajduje się wewnątrz nawiasów klamrowych klasy zewnętrznej.

Prosta klasa wewnętrzna:

```
class MojaKlasaZewnętrzna {  
  
    class MojaKlasaWewnętrzna {  
        void doDziela() {  
            }  
        }  
    }  
}
```

Klasa wewnętrzna jest całkowicie zawarta w klasie zewnętrznej

Klasa wewnętrzna może używać wszystkich składowych i metod klasy zewnętrznej, nawet tych prywatnych.

Klasa wewnętrzna może się posługiwać składowymi i metodami klasy zewnętrznej jak gdyby były zadeklarowane w niej samej.

Klasa wewnętrzna otrzymuje specjalną przepustkę pozwalającą na korzystanie ze składowych i metod klasy zewnętrznej. *Nawet tych składowych i metod, które są prywatne.* Klasa wewnętrzna może się nimi tak posługiwać, jak gdyby były one zdefiniowane bezpośrednio w niej samej. I właśnie to sprawia, że klasy wewnętrzne są tak przydatne i wygodne — dysponują wszystkimi zaletami, jakie dają klasy, a jednocześnie mają specjalne prawa dostępu.

Klasa wewnętrzna używająca składowej klasy zewnętrznej

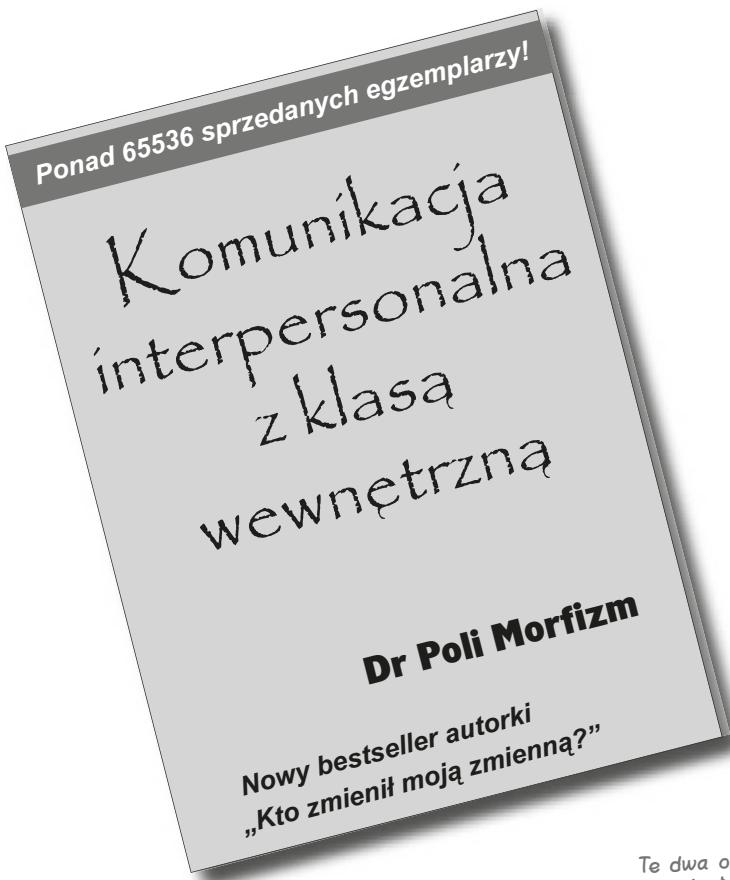
```
class MojaKlasaZewnętrzna {  
    private int x;  
  
    class MojaKlasaWewnętrzna {  
        void doDziela() {  
            x = 42; ← Składowa „x” jest używana, jak gdyby była składową klasy wewnętrznej!  
        }  
    } // koniec klasy wewnętrznej  
}
```

Obiekt klasy wewnętrznej musi być związany z obiektem klasy zewnętrznej*

Pamiętaj, że mówiąc o klasie wewnętrznej, która operuje na czymś w klasie zewnętrznej, tak naprawdę mamy na myśli *obiekt* klasy wewnętrznej, który posługuje się składowymi lub metodami *obiektu* klasy zewnętrznej. Ale *jakiego* obiektu?

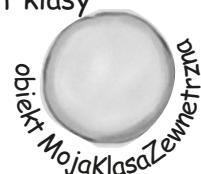
Czy dowolny obiekt klasy wewnętrznej może posługiwać się składowymi i metodami *dowolnego* obiektu klasy zewnętrznej? **Absolutnie nie!**

Obiekt klasy wewnętrznej musi być związany z konkretnym obiektem klasy zewnętrznej przechowywanym na stercie.

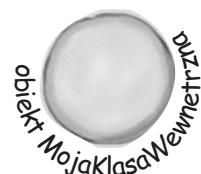


Te dwa obiekty przebywające na stercie łączy szczególna więź.
Obiekt wewnętrzny może korzystać ze składowych obiektu zewnętrznego
(i na odwrót).

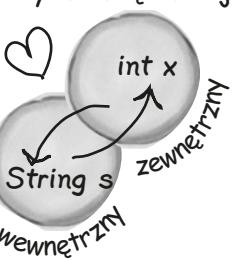
Obiekt wewnętrzny oraz obiekt zewnętrzny są ze sobą powiązane w szczególny sposób.



- 1 Utwórz obiekt klasy zewnętrznej.



- 2 Utwórz obiekt klasy wewnętrznej, używając przy tym obiektu klasy zewnętrznej.



- 3 Teraz obiekt klasy zewnętrznej oraz obiekt klasy wewnętrznej są ze sobą połączone w szczególny sposób.

* Istnieje jeden wyjątek od tej zasady dotyczący bardzo szczególnej sytuacji — klasy wewnętrznej zdefiniowanej w metodzie statycznej. Nie będziemy analizować tej sytuacji i może się zdarzyć, że nigdy w całej swojej karierze nie napotkasz takiego rozwiązania.

Obiekty klasy wewnętrznej

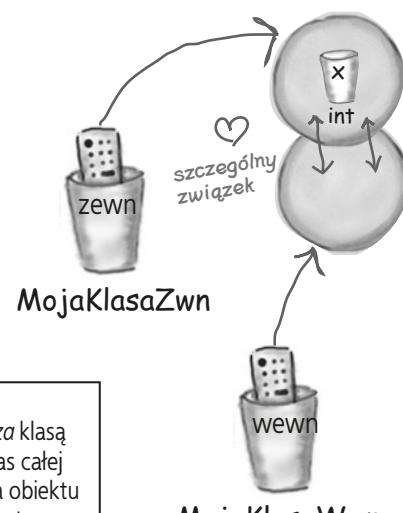
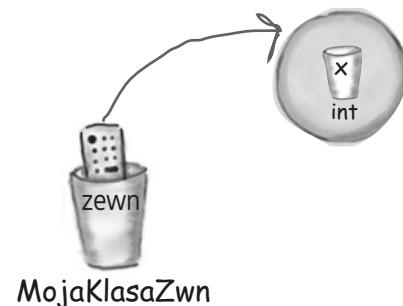
Jak stworzyć obiekt klasy wewnętrznej?

Jeśli tworzysz obiekt klasy wewnętrznej z poziomu kodu należącego od klasy zewnętrznej, to nowy obiekt wewnętrzny zostanie połączony z użyтыm obiektem zewnętrznym. Na przykład, jeśli to kod jakiejś metody utworzy obiekt klasy wewnętrznej, to obiekt ten zostanie połączony z obiektem użyтыm do wywołania tej metody.

Kod klasy zewnętrznej może utworzyć obiekt jednej ze swoich klas wewnętrznych w dokładnie taki sam sposób, w jaki tworzone są obiekty wszelkich innych klas, czyli `new MojaKlasaWewnętrzna()`.

```
class MojaKlasaZwn {  
    private int x; ← Klasa zewnętrzna ma składową  
    prywatną o nazwie „x”.  
    MojaKlasaWwn wewn = new MojaKlasaWwn(); ← Utworzenie obiektu  
    klasy wewnętrznej.  
  
    public void zrobCos() {  
        wewn.doRoboty(); ← Wywołanie metody  
        klasy wewnętrznej.  
    }  
}
```

```
class MojaKlasaWwn {  
    void doRoboty() {  
        x = 42; ← Metoda klasy wewnętrznej używa  
        składowej „x” klasy zewnętrznej, jak  
        gdyby była to jej własna składowa.  
    } // koniec klasy wewnętrznej  
} // koniec klasy zewnętrznej
```



Na marginesie

Istnieje możliwość utworzenia obiektu klasy wewnętrznej z poziomu kodu działającego poza klasą zewnętrzną, jednak należy w tym celu użyć specjalnej składni. Może się zdarzyć, że podczas całej swojej kariery programistycznej nie znajdziesz się w sytuacji, która zmusi Cię do stworzenia obiektu klasy wewnętrznej spoza obiektu klasy zewnętrznej. Jednak na wszelki wypadek, gdyby Cię to interesowało...

```
class Tmp {  
    public static void main(String[] args) {  
        MojaKlasaZwn objZewn = new MojaKlasaZwn();  
        MojaKlasaZwn.MojaKlasaWwn objWewn = objZewn.new MojaKlasaWwn();  
    }  
}
```

Teraz możemy uruchomić kod używający dwóch przycisków

```

public class DwaPrzyciski {
    JFrame ramka;
    JLabel etykieta;

    public static void main(String[] args) {
        DwaPrzyciski gui = new DwaPrzyciski();
        gui.doRoboty();
    }

    public void doRoboty() {
        ramka = new JFrame();
        ramka.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton przyciskEtyk = new JButton("Zmień etykietę");
        przyciskEtyk.addActionListener(new EtykietaListener());
        JButton przyciskKolor = new JButton("Zmień kolor");
        przyciskKolor.addActionListener(new KolorListener());

        etykieta = new JLabel("Oto etykieta");
        MojPanelRysunkowy panel = new MojPanelRysunkowy();

        ramka.getContentPane().add(BorderLayout.SOUTH, przyciskKolor);
        ramka.getContentPane().add(BorderLayout.CENTER, panel);
        ramka.getContentPane().add(BorderLayout.EAST, przyciskEtyk);
        ramka.getContentPane().add(BorderLayout.WEST, etykieta);

        ramka.setSize(400,300);
        ramka.setVisible(true);
    }
}

class EtykietaListener implements ActionListener {
    public void actionPerformed(ActionEvent zdarzenie) {
        etykieta.setText("Auuuuuu!");
    }
} // koniec klasy wewnętrznej
}

class KolorListener implements ActionListener {
    public void actionPerformed(ActionEvent zdarzenie) {
        ramka.repaint();
    }
} // koniec klasy wewnętrznej

```

Główna klasa definiująca interfejs użytkownika nie implementuje już interfejsu ActionListener.

Aktualnie, zamiast odwołania this, do metody rejestrującej odbiorcę zdarzeń przekazywany jest obiekt odpowiedniej klasy wewnętrznej.

Klasa wewnętrzna wie o składowej „etykieta”.

Klasa wewnętrzna może postępować się składową „ramka” bez konieczności jawnego używania odwołania do obiektu klasy zewnętrznej.



Teraz w jednej klasie możemy mieć dwóch odbiorców zdarzeń implementujących interfejs ActionListener.



Java bez tajemnic

Temat wywiadu w tym tygodniu: Obiekt klasy wewnętrznej

Autorzy: Co sprawia, że klasy wewnętrzne są ważne?

Obiekt wewnętrzny: Od czego mam zacząć? Dajemy wam szansę kilkukrotnego zaimplementowania tego samego interfejsu w jednej klasie. Pamiętacie, że w normalnej klasie Javy nie można zaimplementować metody więcej niż jeden raz. Jednak, stosując klasy *wewnętrzne*, każda z nich może implementować ten sam interfejs, dzięki czemu można stworzyć *różne* implementacje tej samej metody interfejsu.

Autorzy: A dlaczego moglibyśmy *chcieć* kilkukrotnie implementować te same metody?

Obiekt wewnętrzny: Przyjrzyjcie się jeszcze raz procedurom obsługi zdarzeń. Zastanówcie się... jeśli chcecie, aby każdy z *trzech* przycisków działał w inny sposób, możecie użyć *trzech* klas wewnętrznych, z których każda będzie implementować interfejs `ActionListener` — a to oznacza, że każda z tych klas będzie dysponować własną implementacją metody `actionPerformed`.

Autorzy: Zatem procedury obsługi zdarzeń są jedynym powodem stosowania klas wewnętrznych?

Obiekt wewnętrzny: Pewnie, że nie. To tylko najbardziej oczywisty przykład. W każdym przypadku, gdy potrzebujecie osobnej klasy, a jednocześnie chcecie, aby zachowywała się ona jak część *innej* klasy, to klasy wewnętrzne są najlepszym — a czasami nawet *jedynym* — rozwiązaniem.

Autorzy: Ciągle nie do końca to rozumiemy. Jeśli chcesz, aby klasa wewnętrzna *zachowywała* się jakby była częścią klasy wewnętrznej, to po co w ogóle tworzyć odrębną klasę? Czy nie można by po prostu umieścić kodu klasy wewnętrznej w klasie wewnętrznej?

Obiekt wewnętrzny: Przed chwilą *podałem* wam przykład — sytuację, w której możecie zaimplementować interfejs więcej niż jeden raz. Jednak, nawet jeśli nie używasz interfejsów, możecie potrzebować dwóch różnych klas, gdyż klasy te reprezentują dwie różne rzeczy. To jest poprawne programowanie zorientowane obiektowo.

Autorzy: Wow! Skoro już o tym mowa. Myśleliśmy, że ważnymi elementami projektowania zorientowanego obiektowo jest wielokrotne wykorzystywanie kodu oraz łatwość jego utrzymania. Chodzi o to, że jeśli masz dwie klasy, to można je niezależnie modyfikować i używać, natomiast jeśli umieścisz to wszystko w jednej klasie... i tak dalej. Ale w przypadku użycia klasy wewnętrznej w efekcie masz tylko jedną *prawdziwą* klasę, nieprawdaż? Jedynie klasa wewnętrzna nadaje się do wielokrotnego używania i jest całkowicie niezależna od wszelkich pozostałych klas. Klasy wewnętrzne nie w pełni nadają

się do wielokrotnego używania. W rzeczywistości spotkaliśmy się z określeniem, że są one „wielokrotnie bezużyteczne”.

Obiekt wewnętrzny: Tak, to prawda. Klasy wewnętrzne nie nadają się do wielokrotnego stosowania aż tak dobrze jak normalne klasy. W rzeczywistości czasami w ogóle nie nadają się do wielokrotnego stosowania, gdyż są ściśle związane ze składowymi i metodami klasy zewnętrznej. Jednak to...

Autorzy: Co potwierdza to, co powiedzieliśmy. Skoro nie nadają się do wielokrotnego wykorzystania, to po co w ogóle zaprzatać sobie głowę oddzielnymi klasami? Oczywiście chodzi nam o inne powody niż te związane z interfejsami, choć i w tym przypadku wydaje nam się, że nie jest to najprostsze rozwiązanie.

Obiekt wewnętrzny: Jak już powiedziałem — musicie pomyśleć o relacji JEST i o polimorfizmie.

Autorzy: W porządku. Myślimy o tym, gdyż...

Obiekt wewnętrzny: Gdyż może się zdarzyć, że klasa zewnętrzna i wewnętrzna będą musiały sprawdzać *różne* relacje JEST! Zacznijmy od przykładu polimorficznego odbiorcy zdarzeń. Jaki jest zadeklarowany typ argumentu w metodzie rejestrującej odbiorcę zdarzeń generowanych przez przycisk? Innymi słowy, gdybyście zajrzaли do dokumentacji API, to jaki byłby typ *rzeczy* (typ klasy lub interfejsu), którą należy przekazać w wywoaniu metody `addActionListener()`?

Autorzy: Trzeba przekazać odbiorcę zdarzeń. Coś, co implementuje konkretny interfejs odbiorcy, a w tym przypadku jest to interfejs `ActionListener`. Tak, tak... to wszystko wiemy. Do czego zmierzasz?

Obiekt wewnętrzny: Chodzi mi o to, że z polimorficznego punktu widzenia dysponujecie metodą, która wymaga przekazania jednego, konkretnego *typu*. Czegoś, co spełnia relację JEST z interfejsem `ActionListener`. Ale co zrobić w sytuacji — i to jest bardzo ważne — gdy wasza klasa musi spełniać relację JEST z czymś, co jest *klasą*, a nie interfejsem?

Autorzy: A czy nasza klasa wewnętrzna nie mogłaby po prostu *rozszerzać* klas, której musi być częścią? Czy właśnie nie na tym ma polegać tworzenie klas potomnych? Jeśli B jest klasą potomną klasą A, to wszędzie tam, gdzie oczekiwany jest obiekt klasy A, można przekazać także obiekt klasy B. No wiesz... „przekaż obiekt Pies tam, gdzie oczekiwany jest obiekt Zwierze”, i te sprawy...

Obiekt wewnętrzny: Tak! Właśnie o to chodzi! Więc co zrobić w sytuacji, gdy relacja JEST musi być spełniona dla dwóch różnych klas? I to klas, które nie należą do tej samej hierarchii?

Autorzy: Więc, w takiej sytuacji... hmmmm. Dobra, chyba zaczynamy rozumieć. Zawsze można *zaimplementować* więcej niż jeden interfejs, ale *rozszerzać* można tylko *jedną* klasę. Czyli, jeśli chodzi o klasy, relacja JEST może być spełniona tylko z jednym typem.

Obiekt wewnętrzny: Dobra robota! Owszem, obiekt nie może być jednocześnie klasy Pies i JButton. Ale co zrobić, jeśli obiekt Pies musi czasami być obiektem JButton (aby można go było przekazać do metody oczekującej obiektu JButton). W takim przypadku klasa Pies (która rozszerza klasę Zwierze, a więc nie może jednocześnie rozszerzać klasę JButton) może mieć klasę *wewnętrzna*, która dzięki temu, iż rozszerza JButton, może w imieniu klasy Pies działać jako klasa JButton. Zatem wszędzie tam, gdzie wymagany jest obiekt JButton, obiekt Pies może przekazać swój wewnętrzny obiekt JButton. Innymi słowy, zamiast wywołania x.pobierzJButton(this), obiekt Pies może użyć wywołania: x.pobierzJButton(new MojWewnJButton()).

Autorzy: Czy możemy prosić o jakiś prosty i przejrzysty przykład?

Obiekt wewnętrzny: Czy pamiętacie używany wcześniej panel rysunkowy, tę klasę potomną klasy JPanel? W bieżącej postaci jest to zwyczajna klasa, która nie ma nic wspólnego z klasą wewnętrzną. I wszystko jest w porządku, gdyż klasa ta nie musi mieć dostępu do składowych głównej klasy definiującej interfejs użytkownika. Ale co by było, gdyby ten dostęp był konieczny? Co zrobić, gdybyśmy tworzyli animację i nasz panel musiałby otrzymywać współrzędne z aplikacji głównej (założymy na przykład, że współrzędne te byłyby wyznaczane na podstawie innych operacji wykonywanych przez użytkownika przy użyciu komponentów interfejsu graficznego). W takim przypadku, gdybyśmy stworzyli panel jako klasę wewnętrzną, to klasa panelu mogłaby rozszerzać klasę JPanel, natomiast klasa zewnętrzna mogłaby rozszerzać dowolną inną klasę.

Autorzy: Tak, rozumiemy! Poza tym panel rysunkowy nie nadaje się do wielokrotnego użycia w tak dużym stopniu, aby tworzyć go jako niezależną klasę. To, co wyświetla, jest bowiem unikalne dla tej konkretnej aplikacji.

Obiekt wewnętrzny: No! W końcu pojawiście się, o co chodzi!

Autorzy: Dobra. W takim razie możemy przejść do kolejnego zagadnienia — natury *relacji* pomiędzy tobą a obiektem klasy zewnętrznej.

Obiekt wewnętrzny: Co jest z wami, ludzie? Za mało pikantnych ploteczek na powaźniejsze tematy, takie jak polimorfizm?

Autorzy: Ech, nawet nie wiesz, ile ludzie mogą wydać na porządną, pikantną opowieść rodem z brukowca. Zatem ktoś cię tworzy i momentalnie stajesz się związany z obiektem klasy zewnętrznej, tak?

Obiekt wewnętrzny: Tak, to prawda. I faktycznie niektórzy porównują to z aranżowanym małżeństwem. Nikt nam nie mówi, z jakim obiektem zostaniemy związani.

Autorzy: Dobrze, będziemy się zatem trzymać tego porównania z małżeństwem. Czy możesz się zatem *rozwieść* i wziąć ślub z jakimś *innym* obiektem?

Obiekt wewnętrzny: Nie, nasz związek jest na całe życie.

Autorzy: Ale czyje życie? Twoje? Obiektu zewnętrznego? A może was oboja?

Obiekt wewnętrzny: Moje. Nie mogę być związany z żadnym innym obiektem zewnętrznym. Dla mnie jedynym wyjściem jest odśmieczacz.

Autorzy: A co z obiektem zewnętrznym? Czy on może się związać z innym obiektem wewnętrznym?

Obiekt wewnętrzny: Wiedziałem. To właśnie *do tego* cały czas zmierzaliście. Tak, tak. Mój tak zwany „współobiekt” może mieć tyle obiektów wewnętrznych, ile zechce.

Autorzy: To jest jak gdyby seryjna monogamia? Czy też może mieć te wszystkie obiekty wewnętrzne jednocześnie?

Obiekt wewnętrzny: Wszystkie jednocześnie. Co, jesteście zadowoleni?

Autorzy: Cóż, to ma sens. I nie zapominaj, że to właśnie *ty* wychwalałeś zalety „wielokrotnej implementacji tego samego interfejsu”. To całkiem sensowne, jeśli bowiem klasa zewnętrzna ma trzy przyciski, to do obsługi zdarzeń będzie potrzebować trzech różnych klas wewnętrznych (a zatem także trzech obiektów tych klas). Dzięki za wszystko. Proszę, tu masz chusteczkę.

Myśli, że ma szczęście, posiadając jednocześnie dwa obiekty klas wewnętrznych. Ale nie wie, że mamy dostęp do wszystkich jego danych prywatnych, wyobraźcie sobie, ile problemów możemy mu przysporzyć...



Wykorzystanie klas wewnętrznych do tworzenia animacji

Przekonaliśmy się, dlaczego klasy wewnętrzne ułatwiają tworzenie odbiorców zdarzeń — gdyż dzięki nim można kilkukrotnie implementować tę samą metodę obsługującą zdarzenia. Jednak teraz przekonamy się, jak przydatne są klasy potomne w sytuacjach, gdy rozszerzają inną klasę niż ich klasa zewnętrzna. Innymi słowy, gdy klasa zewnętrzna i wewnętrzna należą do innych drzew dziedziczenia.

Naszym celem jest stworzenie prostej animacji, która przesuwa kółko w poprzek okna, od jego lewego górnego do prawego dolnego rogu.



Jak proste jest działanie animacji?

- 1 Narysuj obiekt w punkcie o określonych współrzędnych x i y.

g.fillRect(20, 50, 100, 100);

→ 20 pikseli od lewej
50 pikseli od góry

- 2 Powtórznie narysuj obiekt w punkcie o innych współrzędnych x i y.

g.fillRect(25, 55, 100, 100);

→ 25 pikseli od lewej, 55 pikseli od góry
(obiekt został przesunięty nieco w prawo i nieco ku dołowi)

- 3 Powtarzaj poprzedni krok polegający na zmianie współrzędnych rysowanego obiektu tak długo, jak długo ma działać animacja.

Nie istnieja
grupie pytania

P: Dlaczego uczymy się teraz o animacji? Nie przypuszczam, żebym kiedyś miał tworzyć gry.

O: Być może nie będziesz tworzyć gier, ale możesz tworzyć symulacje, których pewne elementy zmieniają się wraz z upływem czasu, prezentując rezultaty działania procesu. Ewentualnie możesz tworzyć narzędzie służące do wizualizacji, które, na przykład, aktualizuje grafikę, pokazując, ile pamięci aktualnie wykorzystuje program, albo jaki jest ruch na serwerze wyrównującym obciążenie. Lub jakkolwiek inny program, który musi odczytywać zbiór nieustannie zmieniających się liczb i zamieniać je na coś, co można wykorzystać do uzyskania pewnych informacji na podstawie tych liczb.

Czy to wszystko nie brzmi jak gdyby miało związek z prowadzeniem biznesu? Oczywiście, to jest tylko „oficjalne usprawiedliwienie”. Prawdziwym powodem, dla którego zajmujemy się zagadnieniem animacji, jest to, iż stanowi ona prosty sposób przedstawienia innych metod wykorzystania klas wewnętrznych. (Oraz ponieważ po prostu lubimy animacje, a nasza kolejna książka będzie poświęcona Java 2 Enterprise Edition i wiemy, że tam nie znajdziemy miejsca na animacje).

Tak naprawdę, to chcemy czegoś takiego...

```
class MojPanelRysunkowy extends JPanel {  
    public void paintComponent(Graphics g) {  
        g.setColor(Color.orange);  
        g.fillOval(x,y,100,100);  
    }  
}
```

Każde wywołanie metody
paintComponent()
spowoduje wyświetlenie
owalu w innym miejscu.



Zaostrz ołówek

Jednak jak będą określane współrzędne x i y?

Kto wywoła metodę repaint()?

Sprawdź, czy będziesz w stanie **zaprojektować proste rozwiązanie**, które pozwoli stworzyć animację przesuwającą owal z lewego górnego do prawego dolnego wierzchołka panelu. Nasze rozwiązanie zostało przedstawione na następnej stronie, więc nie przewracaj kartki, póki nie rozwiążesz tego ćwiczenia!

Duża podpowiedź: Stwórz panel rysunkowy jako klasę wewnętrzną.

Kolejna podpowiedź: Nie umieszczaj jakichkolwiek pętli w metodzie paintComponent().

Poniżej zapisz swoje pomysły (lub gotowy kod):

Animacja wykorzystująca klasę wewnętrzną

Pełny kod prostej animacji

```
import javax.swing.*;
import java.awt.*;

public class ProstaAnimacja {
    int x = 70; } ← W głównej klasie definiującej interfejs
    int y = 70; ← użytkownika tworzymy dwie składowe,
                  określające współrzędne x i y punktu,
                  w którym zostanie wyświetcone kółko.

    public static void main(String[] args) {
        ProstaAnimacja gui = new ProstaAnimacja();
        gui.doRoboty();
    }

    public void doRoboty() {
        JFrame ramka = new JFrame();
        ramka.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        MojPanelRysunkowy panel = new MojPanelRysunkowy(); ← Tu nie ma nic nowego.
        ramka.getContentPane().add(panel); ← Tworzymy komponenty
        ramka.setSize(300,300); ← i umieszczamy je w ramce.
        ramka.setVisible(true);
    }

    for (int i = 0; i < 130; i++) { ← Powtarzamy pętlę 130 razy.

        x++; ← Inkrementujemy
        y++; ← współrzędne x i y.

        panel.repaint(); ← Nakazujemy panelowi, aby odświeżyć
                           swoją zawartość (dzięki czemu będziemy
                           mogli wyświetlić kółko w nowym miejscu).

        try {
            Thread.sleep(50); ← Zwalniamy trochę działanie animacji (gdybyśmy tego
                               nie zrobili, w ogóle nie byłbyś w stanie ZOBACZYĆ
                               tego efektu). Nie przejmuj się, nie oczekiwaliśmy,
                               że będziesz o tym wiedział. Tym zagadnieniem
                               zajmiemy się w rozdziale 15.
        } catch (Exception ex) { }
    }
}

} // koniec doRoboty()

class MojPanelRysunkowy extends JPanel {
    public void paintComponent(Graphics g) {
        g.setColor(Color.green);
        g.fillOval(x,y,40,40); ← Wykorzystujemy wciąż modyfikowane
                               współrzędne z klasy zewnętrznej.
    }
} // koniec klasy wewnętrznej
} // koniec klasy zewnętrznej
```

W tym momencie zaczyna się prawdziwa akcja!

Teraz panel jest klasą wewnętrzną.

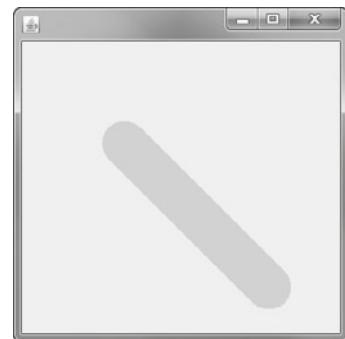
Och jej! Kółko się nie przesuwa... Ono się rozmazuje!

Co zrobiliśmy nie tak?

Jest niewielki błąd w kodzie metody `paintComponent()`.

Zapomnieliśmy o usunięciu kółka, które już było wyświetcone! I dlatego zostaje po nim ślad.

Jedyną rzeczą, jaką musimy zrobić, aby rozwiązać ten problem, jest wypełnienie całego panelu kolorem tła bezpośrednio przed narysowaniem kółka. W poniższym przykładzie na samym początku metody `paintComponent()` zostały dodane dwa wiersze kodu: pierwszy z nich określa, że wybranym kolorem ma być biały (kolor tła panelu), a drugi wypełnia cały prostokąt panelu wybranym kolorem. Mówiąc po polsku, poniższy kod informuje, że: „należy wypełnić prostokąt, zaczynając od punktu 0,0 (0 pikseli od lewej i 0 pikseli od góry), a szerokość i wysokość wypełnianego obszaru ma odpowiadać bieżącej szerokości i wysokości panelu”.



To nie wygląda tak, jak chcieliśmy.

```
public void paintComponent(Graphics g) {
    g.setColor(Color.white);
    g.fillRect(0,0,this.getWidth(), this.getHeight());
    g.setColor(Color.green);
    g.fillOval(x,y,40,40);
}
```

getWidth() oraz getHeight() to metody odziedziczone po klasie JPanel.



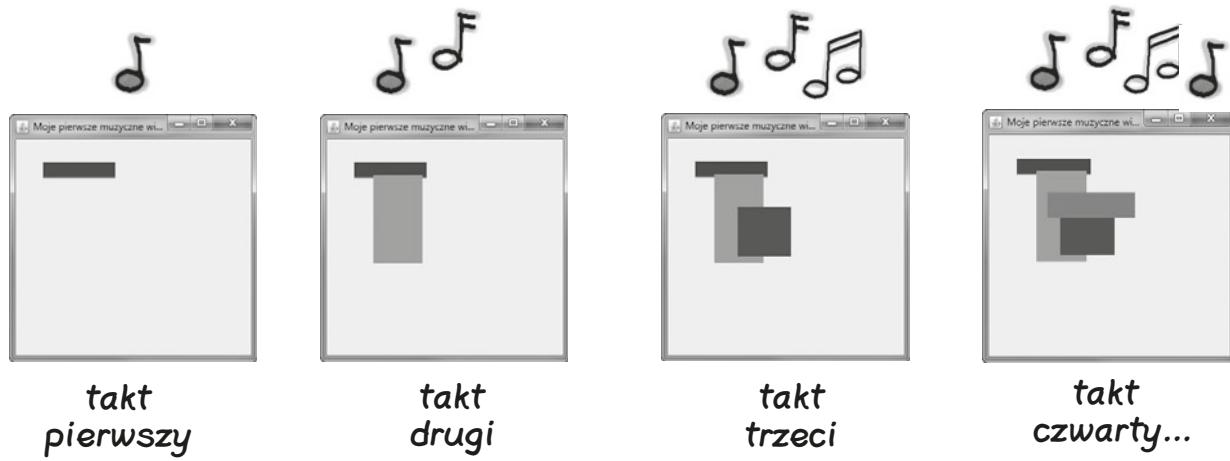
Zaostrz ołówek (opcjonalne, dla zabawy)

W jaki sposób zmieniałbyś współrzędne x i y punktu, w którym jest wyświetlone kółko, aby stworzyć przedstawione poniżej animacje? (Załóż, że w pierwszym przykładzie przyrosty współrzędnych punktu wynoszą 3 piksele).

1			X <u>+3</u>	Y <u>+3</u>
początek koniec				
2			X <u> </u>	Y <u> </u>
początek koniec				
3			X <u> </u>	Y <u> </u>
początek koniec				

1			X <u> </u>	Y <u> </u>
początek koniec				
2			X <u> </u>	Y <u> </u>
początek koniec				
3			X <u> </u>	Y <u> </u>
początek koniec				

Kod od kuchni



Stwórzmy „muzyczne wideo”. Użyjemy losowo generowanej grafiki, która będzie generowana w takt muzyki.

Przy okazji zarejestrujemy się i będziemy odbierać nowy rodzaj zdarzeń, które nie mają nic wspólnego z graficznym interfejsem użytkownika – zdarzenie generowane przez samą muzykę.

Pamiętaj, że ta część rozdziału jest całkowicie opcjonalna. Jednak uważamy, że jej przeczytanie może Ci się przydać. Poza tym pewnie Ci się spodoba. I możesz jej użyć, żeby zrobić wrażenie na innych.

(W porządku, zrobi wrażenie tylko na osobach, na których łatwo wywrzeć wrażenie, ale pomimo to...)

Odbieranie zdarzeń niezwiązanych z interfejsem użytkownika

No dobrze, może to nie będzie muzyczne wideo, ale na pewno program, który rysuje na ekranie losowe figury w takt muzyki. Najogólniej rzecz biorąc, program „słucha” taktów muzyki i w odpowiedzi na każdy z nich rysuje prostokąt o losowych wymiarach i położeniu.

To zadanie stawia przed nami nowe wyzwania. Jak do tej pory odbieraliśmy jedynie zdarzenia związane z graficznym interfejsem użytkownika, jednak teraz musimy odbierać szczególny rodzaj zdarzeń MIDI. Okazuje się, że odbieranie zdarzeń niezwiązańych z interfejsem użytkownika oraz związanych z nim jest bardzo podobne — w obu przypadkach należy zaimplementować interfejs odbiorcy, zarejestrować odbiorcę w źródle zdarzeń, a następnie siąść spokojnie i czekać, aż źródło wywoła metodę obsługującą zdarzenia (zdefiniowaną w interfejsie odbiorcy).

Najprostszym sposobem „słuchania” taktów muzyki byłoby zarejestrowanie odbiorcy i odbieranie faktycznych zdarzeń MIDI; dzięki temu, gdy tylko sekwenser odebrałby zdarzenie, odebrałby je także nasz kod, który w odpowiedzi mógłby coś narysować. Jednak... jest pewien problem. A właściwie to błąd, który nie pozwala na odbieranie zdarzeń MIDI, które *sam* generujemy (a konkretnie chodzi o zdarzenia NOTE ON).

A zatem musimy zastosować inne rozwiązanie. Istnieje inny typ zdarzeń MIDI, które możemy odbierać. Są to zdarzenia ControllerEvent. Nasze rozwiązanie będzie polegało na zarejestrowaniu odbiorcy tych zdarzeń i zagwarantowaniu, że dla każdego zdarzenia NOTE ON, w tym samym taktie, zostanie wygenerowane zdarzenie ControllerEvent. Jak możemy zagwarantować, że w tym samym czasie zostanie wygenerowane zdarzenie ControllerEvent? Należy je dodać do ścieżki, tak samo jak wszystkie inne zdarzenia! Innymi słowy, nasza sekwencja muzyki powinna wyglądać następująco:

Takt 1 - NOTE ON, CONTROLLER EVENT.

Takt 2 - NOTE OFF.

Takt 3 - NOTE ON, CONTROLLER EVENT.

Takt 4 - NOTE OFF.

I tak dalej.

Zanim jednak przedstawimy pełny kod programu, uproścmy sobie nieco zadanie tworzenia i dodawania komunikatów — zdarzeń MIDI, gdyż w tym programie będziemy ich tworzyć znacznie więcej.

Co nasz artystyczny program muzyczny musi robić?

- 1 Tworzyć serię komunikatów-zdarzeń MIDI, aby grać losowe nuty na pianinie (albo jakimkolwiek innym instrumentem).
- 2 Rejestrować odbiorcę zdarzeń.
- 3 Uruchamiać odtwarzanie muzyki przez sekwenser.
- 4 Za każdym razem, gdy zostanie wywołana metoda odbiorcy zdarzeń, rysować na panelu losowy prostokąt i wywoływać metodę repaint().

Program stworzymy w trzech krokach:

- 1 Wersja pierwsza — zawiera kod ułatwiający tworzenie i dodawanie zdarzeń MIDI, co okaże się niezmiernie przydatne, gdyż będziemy ich potrzebowali bardzo dużo.
- 2 Wersja druga — rejestruje odbiorców i odbiera zdarzenia, jednak nie dysponuje interfejsem graficznym. Dla każdego taktu wyświetla w wierszu poleceń odpowiedni komunikat.
- 3 Wersja trzecia — to jest to — druga wersja programu wyposażona w interfejs graficzny.

Prostszy sposób tworzenia komunikatów–zdarzeń

Jak na razie tworzenie komunikatów oraz zdarzeń i umieszczanie ich na ścieżce jest dosyć męczące. Dla każdego komunikatu musimy stworzyć odpowiedni obiekt (w naszym przypadku jest to obiekt ShortMessage), wywołać metodę `setMessage()`, utworzyć obiekt zdarzenia (MidiEvent) dla danego komunikatu i dodać go do ścieżki. W kodzie przedstawionym w poprzednim rozdziale dla każdego komunikatu wykonywaliśmy wszystkie wymienione czynności. A to oznacza osiem wierszy kodu tylko po to, aby rozpocząć i skończyć granie jednej nuty! Cztery wiersze, aby dodać zdarzenie NOTE ON i kolejne cztery, aby dodać zdarzenie NOTE OFF.

```
ShortMessage a = new ShortMessage();
a.setMessage(144, 1, nuta, 100);
MidiEvent nutaP = new MidiEvent(a, 1);
sciezka.add(nutaP);
```

```
ShortMessage b = new ShortMessage();
b.setMessage(128, 1, nuta, 100);
MidiEvent nutaK = new MidiEvent(b, 16);
sciezka.add(nutaK);
```

Stwórzmy statyczną metodę pomocniczą, która będzie tworzyć komunikat i zwracać obiekt MidiEvent

Cztery argumenty przekazujące dane dotyczące komunikatu.

Takt zdarzenia określający, KIEDY ten komunikat powinien zostać zgłoszony.

```
public MidiEvent tworzZdarzenie(int plc, int kanal, int jeden, int dwa, int takt) {
    MidiEvent zdarzenie = null;
    try {
        ShortMessage a = new ShortMessage();
        a.setMessage(plc, kanal, jeden, dwa);
        zdarzenie = new MidiEvent(a, takt);
    } catch(Exception e) { }
    return zdarzenie;
}
```

O rany! Metoda z pięcioma parametrami.

Tworzymy komunikat i obiekt zdarzenia, wykorzystując przy tym parametry metody.

Zwracamy zdarzenie (obiekt MidiEvent) zawierający utworzony komunikat.

Przykład użycia nowej statycznej metody tworZdarzenie()

W poniższym programie nie ma żadnej obsługi zdarzeń ani graficznego interfejsu użytkownika; program tworzy jedynie sekwencję piętnastu coraz wyższych nut. Ma Ci pokazać, w jaki sposób należy posługiwać się naszą nową metodą `tworZdarzenie()`. Dzięki tej metodzie kod pozostałych dwóch wersji programu odtwarzacza będzie znacznie krótszy i prostszy.

```
import javax.sound.midi.*; ← Nie zapomnij o importie
class Miniodtwarzacz1 {

    public static void main(String[] args) {

        try {

            Sequencer sekvenser = MidiSystem.getSequencer(); ← Tworzymy (i otwieramy) sekvenser.
            sekvenser.open(); ← Tworzymy sekwencję i ścieżkę.

            Sequence sekw = new Sequence(Sequence.PPQ, 4); ← Tworzymy grupę zdarzeń, które pozwala
            Track sciezka = sekw.createTrack(); ← nam odegrać nuty (odpowiadające klawiszom
                                                fortepianu o numerach od 5 do 61).

            for (int i = 5; i < 61; i+=4) { ← Wywołujemy naszą nową metodę tworZdarzenie()
                sciezka.add(tworZdarzenie(144,1,i,100,i));
                sciezka.add(tworZdarzenie(128,1,i,100,i + 2)); ← w celu stworzenia komunikatu i zdarzenia,
            } // koniec pętli                                     następnie uzyskany wynik (obiekt MidiEvent
                                                       zwrócony przez metodę tworZdarzenie()) dodajemy
                                                       do ścieżki. Te dwa wiersze kodu tworzą parę
                                                       zdarzeń NOTE ON (144) oraz NOTE OFF (128).

            sekvenser.setSequence(sekw); } Uruchamiamy odtwarzanie.
            sekvenser.setTempoInBPM(220);
            sekvenser.start(); }

        } catch (Exception ex) { ex.printStackTrace(); }
    } // koniec metody

    public static MidiEvent tworZdarzenie(int plc, int kanal, int jeden, int dwa, int takt) {
        MidiEvent zdarzenie = null;
        try {
            ShortMessage a = new ShortMessage();
            a.setMessage(plc, kanal, jeden, dwa);
            zdarzenie = new MidiEvent(a, takt);

        } catch(Exception e) { }
        return zdarzenie;
    }
} // koniec klasy
```

Wersja druga — rejestracja odbiorcy i odbieranie zdarzeń ControllerEvent

```

import javax.sound.midi.*;
public class Miniodtwarzacz2 implements ControllerEventListener {
    public static void main(String[] args) {
        Miniodtwarzacz2 mini = new Miniodtwarzacz2();
        mini.doRoboty();
    }

    public void doRoboty() {
        try {
            Sequencer sekwenser = MidiSystem.getSequencer();
            sekwenser.open();

            int[] zdarzeniaObslugiwane = {127};
            sekwenser.addControllerEventListener(this, zdarzeniaObslugiwane);

            Sequence sekw = new Sequence(Sequence.PPQ, 4);
            Track sciezka = sekw.createTrack();

            for (int i = 5; i < 61; i+=4) {
                sciezka.add(tworzZdarzenie(144,1,i,100,i));
                sciezka.add(tworzZdarzenie(176,1,127,0,i)); ←
                sciezka.add(tworzZdarzenie(128,1,i,100,i + 2));
            } // koniec pętli

            sekwenser.setSequence(sekw);
            sekwenser.setTempoInBPM(220);
            sekwenser.start();
        } catch (Exception ex) { ex.printStackTrace(); }
    } // koniec

    public void controlChange(ShortMessage zdarzenie) {
        System.out.println("la");
    } ←
}

public static MidiEvent tworzZdarzenie(int plc, int kanal, int jeden, int dwa, int takt) {
    MidiEvent zdarzenie = null;
    try {
        ShortMessage a = new ShortMessage();
        a.setMessage(plc, kanal, jeden, dwa);
        zdarzenie = new MidiEvent(a, takt);
    } catch(Exception e) {} ←
    return zdarzenie;
}
} // koniec klasy

```

Musimy odbierać zdarzenia ControllerEvent, zatem implementujemy odpowiedni interfejs.

Rejestrujemy odbiorcę zdarzeń w sekwenserze. Metoda rejestrująca wymaga przekazania odbiorcy zdarzeń ORAŻ tablicy liczb całkowitych typu int reprezentującej listę zdarzeń ControllerEvent, które chcemy odbierać. Nas interesują jedynie zdarzenia o numerze 127.

Oto jak zapewniamy sobie możliwość odbierania informacji o takach — dodajemy do ścieżki nasze WŁASNE zdarzenie ControllerEvent (numer 176 określa, że dane zdarzenie jest typu ControllerEvent), używając przy tym argumentu, który przypisuje temu zdarzeniu numer 127. To zdarzenie nie powoduje wykonania JAKICHKOLWIEK operacji! Umieszczamy je na ścieżce tylko po to, abyśmy mogli odebrać zdarzenie za każdym razem, gdy zostanie zagrana jakaś nuta. Innymi słowy, są one tworzone wyłącznie po to, abyśmy mogli odbierać jakieś zdarzenia (nie możemy bowiem odbierać zdarzeń NOTE ON ani NOTE OFF). Zauważ, że zdarzenia te są zgłaszane w tym samym takcie co zdarzenia NOTE ON. Dzięki temu będziemy wiedzieć, kiedy zostanie zgłoszone zdarzenie NOTE ON, gdyż w tym samym czasie pojawi się także NASZE zdarzenie.

Metoda obsługująca zdarzenia (należąca do interfejsu związanego ze zdarzeniami ControllerEvent). Za każdym razem, gdy odbierzemy zdarzenie, w wierszu polecień wyświetlimy ciąg znaków „la”.

Fragmenty kodu, które różnią się od poprzedniej wersji programu, zostały przedstawione na szarym tle. (Poza tym w tej wersji większość operacji została usunięta z metody main()).

Wersja trzecia — rysowanie grafiki w takt muzyki

Ta ostateczna wersja programu bazuje na wersji drugiej, dodając do niej interfejs graficzny. Program tworzy ramkę i dodaje do niej panel, a za każdym razem, gdy zostanie przechwycone zdarzenie, rysuje nowy prostokąt i odświeża ekran. W porównaniu z poprzednią wersją programu wprowadzono jeszcze jedną zmianę, dotyczy ona nut, które teraz są wybierane losowo, a nie, jak było wcześniej, rozmieszczone coraz wyżej na skali.

Najważniejszą modyfikacją wprowadzoną w kodzie programu (oczywiście oprócz stworzenia prostego interfejsu graficznego) jest zaimplementowanie interfejsu ControllerEventListener w panelu, a nie w głównej klasie programu. Dzięki temu, gdy panel (czyli klasa wewnętrzna) odbierze zdarzenie, będzie wiedział, co zrobić, aby narysować prostokąt.

Pełny kod tej wersji programu został przedstawiony na następnej stronie.

Klasa wewnętrzna panelu rysunkowego:

```
class MojPanelGraf extends JPanel implements ControllerEventListener {
    boolean komunikat = false; // Przypisujemy flagę wartość false i zmienimy ją na true jedynie wtedy, gdy odbierzemy zdarzenie.

    public void controlChange(ShortMessage zdarzenie) {
        komunikat = true; // Odebraliśmy zdarzenie, a zatem przypisujemy flagę wartość true i wywołujemy metodę repaint().
        repaint();
    }

    public void paintComponent(Graphics g) {
        if (komunikat) { // Musimy wykorzystać flagę, gdyż także INNE czynniki mogą spowodować wywołanie metody repaint(), a chcemy odświeżyć zawartość panelu wyłącznie w przypadku odebrania zdarzenia.
            Graphics2D g2 = (Graphics2D) g;
            int c = (int) (Math.random() * 250);
            int z = (int) (Math.random() * 250);
            int n = (int) (Math.random() * 250);
            g.setColor(new Color(c,z,n)); // Pozostała część kodu odpowiada za wybór losowego koloru i narysowanie pseudolosowego prostokąta.

            int wys = (int) ((Math.random() * 120) + 10);
            int szer = (int) ((Math.random() * 120) + 10);
            int x = (int) ((Math.random() * 40) + 10);
            int y = (int) ((Math.random() * 40) + 10);
            g.fillRect(x,y,wys,szer);
            komunikat = false;
        }
    }
}
```

Panel jest odbiorcą zdarzeń.

Przypisujemy flagę wartość false i zmienimy ją na true jedynie wtedy, gdy odbierzemy zdarzenie.

Odebraliśmy zdarzenie, a zatem przypisujemy flagę wartość true i wywołujemy metodę repaint().

Musimy wykorzystać flagę, gdyż także INNE czynniki mogą spowodować wywołanie metody repaint(), a chcemy odświeżyć zawartość panelu wyłącznie w przypadku odebrania zdarzenia.

Pozostała część kodu odpowiada za wybór losowego koloru i narysowanie pseudolosowego prostokąta.

Kod programu Miniodtwarzacz3

```
import javax.sound.midi.*;
import javax.swing.*;
import java.awt.*;

class Miniodtwarzacz3 {

    static JFrame ramka = new JFrame("Moje pierwsze muzyczne wideo");
    static MojPanelGraf panel;

    public static void main(String[] args) {
        Miniodtwarzacz3 mini = new Miniodtwarzacz3();
        mini.doRoboty();
    } // koniec metody

    public void konfigurujGUI() {
        ramka.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        panel = new MojPanelGraf();
        ramka.setContentPane(panel);
        ramka.setBounds(30,30,300,300);
        ramka.setVisible(true);
    } // koniec metody

    public void doRoboty() {
        konfigurujGUI();

        try {

            Sequencer sekvenser = MidiSystem.getSequencer();
            sekvenser.open();
            sekvenser.addControllerEventListener(panel, new int[] {127});
            Sequence sekw = new Sequence(Sequence.PPQ, 4);
            Track sciezka = sekw.createTrack();

            int r = 0;
            for (int i = 5; i < 60; i+=4) {

                r = (int) ((Math.random() * 50) + 1);
                sciezka.add(tworzZdarzenie(144,1,r,100,i));
                sciezka.add(tworzZdarzenie(176,1,127,0,i));
                sciezka.add(tworzZdarzenie(128,1,r,100,i + 2));
            } // koniec pętli

            sekvenser.setSequence(sekw);
            sekvenser.setTempoInBPM(220);
            sekvenser.start();
        } catch (Exception ex) { ex.printStackTrace(); }
    } // koniec metody
}
```



Zaostrz ołówek

Oto pełny kod trzeciej wersji programu. Został on stworzony bezpośrednio na bazie drugiej wersji programu. Spróbuj sam opisać kod bez zaglądania do wcześniejszych jego wersji i towarzyszących im opisów.

```
public static MidiEvent tworzZdarzenie(int plc, int kanal, int jeden, int dwa, int takt) {  
    MidiEvent zdarzenie = null;  
    try {  
        ShortMessage a = new ShortMessage();  
        a.setMessage(plc, kanal, jeden, dwa);  
        zdarzenie = new MidiEvent(a, takt);  
  
    } catch(Exception e) { }  
    return zdarzenie;  
} // koniec metody  
  
class MojPanelGraf extends JPanel implements ControllerEventListener {  
    boolean komunikat = false;  
  
    public void controlChange(ShortMessage zdarzenie) {  
        komunikat = true;  
        repaint();  
    }  
  
    public void paintComponent(Graphics g) {  
        if (komunikat) {  
  
            Graphics2D g2 = (Graphics2D) g;  
  
            int c = (int) (Math.random() * 250);  
            int z = (int) (Math.random() * 250);  
            int n = (int) (Math.random() * 250);  
  
            g.setColor(new Color(c,z,n));  
  
            int wys = (int) ((Math.random() * 120) + 10);  
            int szer = (int) ((Math.random() * 120) + 10);  
  
            int x = (int) ((Math.random() * 40) + 10);  
            int y = (int) ((Math.random() * 40) + 10);  
  
            g.fillRect(x,y,wys,szer);  
            komunikat = false;  
        } // koniec if  
    } // koniec metody  
} // koniec klasy wewnętrznej  
} // koniec klasy
```

Ćwiczenie. Zgadnij kim jestem



Ćwiczenie

Kim jestem?



Grupa zamaskowanych komponentów Javy gra w grę towarzyską o nazwie „Zgadnij kim jestem?”. Komponenty dają Ci podpowiedzi, a Ty na ich podstawie starasz się odgadnąć, kim one są. Załóż, że komponenty zawsze mówią prawdę. Jeśli mówią coś, co może być prawdą w odniesieniu do kilku z nich, wybierz wszystkie komponenty, dla których podane stwierdzenie jest prawdziwe. W pustych miejscach obok podanych podpowiedzi podaj nazwy komponentów biorących udział w zabawie.

W dzisiejszej zabawie udział biorą:

Zaprozone zostały wszystkie czarujące osobistości, jakie pojawiły się w tym rozdziale.

W swoich rękach mam cały graficzny interfejs użytkownika aplikacji.

Istnieją dla każdego typu zdarzeń.

Kluczowa metoda odbiorcy zdarzeń.

Ta metoda określa wielkość ramki.

Choć dodajesz kod do tej metody, to jednak sam nigdy jej nie wywołujesz.

Kiedy użytkownik wykona jakąś czynność, to jest to...

Większość z nich to źródła zdarzeń.

Przenoszę dane z powrotem do odbiorcy zdarzeń.

Metoda addXxxListener informuje, że obiekt jest...

Jestem metodą służącą do rejestrowania odbiorców zdarzeń.

W tej metodzie umieszczany jest cały kod tworzący grafikę.

Zazwyczaj jestem związana z konkretnym obiektem.

Jakiego typu jest w rzeczywistości parametr „g” w (Graphics g)?

To metoda, która zapewnia wywołanie metody paintComponent().

Pakiet, w którym znajduje się większość „swingujących” elementów graficznego interfejsu użytkownika.



Ćwiczenie

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class PrzyciskWewnetrzny {

    JFrame ramka;
    JButton p;

    public static void main(String[] args) {
        PrzyciskWewnetrzny gui = new PrzyciskWewnetrzny();
        gui.doRoboty();
    }

    public void doRoboty() {
        ramka = new JFrame();

        ramka.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        p = new JButton("A");
        p.addActionListener();

        ramka.getContentPane().add(BorderLayout.SOUTH, p);
        ramka.setSize(200,100);
        ramka.setVisible(true);
    }

    class PListener extends ActionListener {
        public void actionPerformed(ActionEvent e) {
            if (p.getText().equals("A")) {
                p.setText("B");
            } else {
                p.setText("A");
            }
        }
    }
}

```

BĄDŹ kompilatorem

Kod przedstawionych na tej stronie stanowi niezależny, kompletny plik źródłowy programu napisanego w Javie.



Twoim zadaniem jest stać się kompilatorem i określić, czy program ten można skompilować czy nie. Jeśli nie można, to jak go poprawić? Jeśli można, to jakie wygenerowane zostaną wyniki?

Zagadka. Zagadkowy basen



Zagadkowy basen



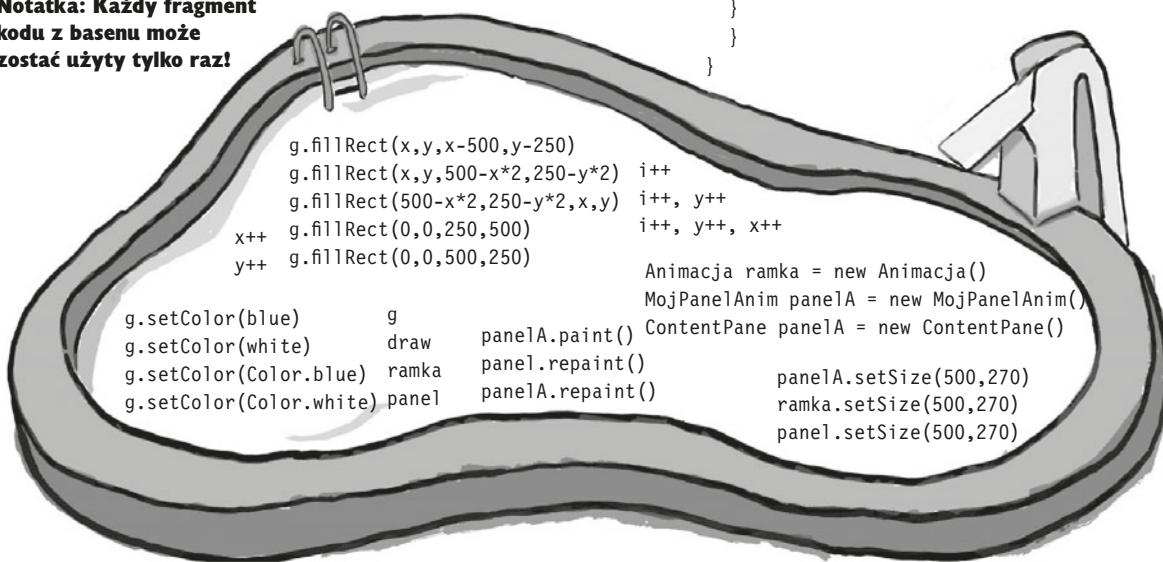
Twoim **zadaniem** jest wybranie fragmentów kodu z basenu i umieszczenie ich w miejscach kodu oznaczonych podkreśleniami. Każdy fragment kodu **możne** być użyty więcej niż raz, a co więcej, nie wszystkie fragmenty zostaną wykorzystane. **Zadanie** polega na stworzeniu klasy, którą będzie można skompilować i która wygeneruje wyniki przedstawione poniżej.

Wyniki

Zadziwiający, zmniejszający się, niebieski prostokąt. Ten program wyświetla niebieski prostokąt, który będzie się coraz bardziej zmniejszać, aż w końcu zupełnie zniknie.



Notatka: Każdy fragment kodu z basenu może zostać użyty tylko raz!



```
import javax.swing.*;
import java.awt.*;
public class Animacja {
    int x = 1;
    int y = 1;
    public static void main(String[] args) {
        Animacja gui = new Animacja();
        gui.doRoboty();
    }
}

private void doRoboty() {
    JFrame ____ = new JFrame();
    ramka.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    _____.getContentPane().add(panelA);
    _____.setVisible(true);
    for (int i = 0; i < 124; _____ ) {
        _____;
        _____;
    }
    try {
        Thread.sleep(50);
    } catch(Exception ex) { }
}
}

public class MojPanelAnim extends JPanel {
    public void paintComponent(Graphics____) {
        _____;
        _____;
        _____;
        _____;
    }
}
}
```



Ćwiczenie Rozwiązanie

Kim jestem?

W swoich rękach mam cały graficzny interfejs użytkownika aplikacji.

Istnieje dla każdego typu zdarzeń.

Kluczowa metoda odbiorcy zdarzeń.

Ta metoda określa wielkość ramki.

Choć dodajesz kod do tej metody, to jednak sam nigdy jej nie wywołujesz.

Kiedy użytkownik wykona jakąś czynność, to jest to...

Większość z nich to źródła zdarzeń.

Przenoszę dane z powrotem do odbiorcy zdarzeń.

Metoda addXxxListener informuje, że obiekt jest...

Jestem metodą służącą do rejestrowania odbiorców zdarzeń.

W tej metodzie umieszczany jest cały kod tworzący grafikę.

Zazwyczaj jestem związana z konkretnym obiektem.

Jakiego typu jest w rzeczywistości parametr „g” w (Graphics g)?

To metoda, która zapewnia wywołanie metody paintComponent().

Pakiet, w którym znajduje się większość „swingujących” elementów graficznego interfejsu użytkownika.

JFrame

interfejs odbiorcy zdarzeń

actionPreformed()

setSize()

paintComponent()

zdarzenie

komponenty biblioteki Swing

obiekt zdarzenia

źródłem zdarzeń

addActionListener()

paintComponent()

klasa wewnętrzna

Graphics2D

repaint()

javax.swing

BĄDŹ kompilatorem

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class PrzyciskWewnętrzny {

    JFrame ramka;
    JButton p;

    public static void main(String[] args) {
        PrzyciskWewnętrzny gui = new PrzyciskWewnętrzny();
        gui.doRoboty();
    }

    public void doRoboty() {
        ramka = new JFrame();
        ramka.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        p = new JButton("A");
        p.addActionListener(new PListener());

        ramka.getContentPane().add(BorderLayout.SOUTH, p);
        ramka.setSize(200,100);
        ramka.setVisible(true);
    }

    class PListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if (p.getText().equals("A")) {
                p.setText("B");
            } else {
                p.setText("A");
            }
        }
    }
}
```

Kiedy błędy w programie zostaną naprawione, będzie on wyświetlał okno zawierające jeden przycisk: klikanie tego przycisku będzie powodować naprzemienne zmianianie jego etykiety z „A” na „B” lub z „B” na „A”.

Metoda addActionListener() wymaga przekazania obiektu klasy, która implementuje interfejs ActionListener.

ActionListener to interfejs, a interfejsy są implementowane, a nie rozszerzane.



Zagadkowy basen

Zadziwiający, zmniejszający się, niebieski prostokąt.



```
import javax.swing.*;
import java.awt.*;

public class Animacja {

    int x = 1;
    int y = 1;

    public static void main(String[] args) {
        Animacja gui = new Animacja();
        gui.doRoboty();
    }

    private void doRoboty() {
        JFrame ramka = new JFrame();
        ramka.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        MojPanelAnim panelA = new MojPanelAnim();
        ramka.getContentPane().add(panelA);
        ramka.setSize(500,270);
        ramka.setVisible(true);
        for (int i = 0; i < 124; i++,x++,y++ ) {
            x++;
            panelA.repaint();
            try {
                Thread.sleep(50);
            } catch(Exception ex) { }
        }
    }

    public class MojPanelAnim extends JPanel {
        public void paintComponent(Graphics g) {
            g.setColor(Color.white);
            g.fillRect(0,0,500,250);
            g.setColor(Color.blue);
            g.fillRect(x,y,500-x*2,250-y*2);
        }
    }
}
```

13. Stosowanie biblioteki Swing

Popracuj nad Swingiem



Dlaczego ta piłka nie
leci tam, gdzie ja chcę?
(Na przykład prosto w twarz
Marysi). Muszę się nauczyć
ją kontrolować.

Swing jest łatwy. Chyba że naprawdę zwracasz uwagę na to, gdzie zostaną wyświetcone poszczególne elementy interfejsu użytkownika. Kod wykorzystujący bibliotekę Swing *wygląda na prosty*, jednak potem go komplujesz, uruchamiasz, patrzysz na wyniki i myślisz: „Hej, to nie miało być w tym miejscu”. Elementem, który sprawia, że tworzony *kod jest prosty*, a *kontrola położenia elementów trudna*, jest **menadżer układu**. Obiekty menadżerów układu kontrolują wielkości i położenie komponentów tworzących graficzny interfejs użytkownika aplikacji. Wykonują one za Ciebie niesamowicie wiele pracy, jednak jej efekty nie zawsze będą Ci się podobać. Na przykład chcesz, aby dwa przyciski miały takie same wymiary, lecz ich wielkości są różne. Chcesz, aby pole tekstowe miało trzy centymetry długości, a jego długość wynosi dziewięć centymetrów. Albo jeden. I jest wyświetlone *poniżej* etykiety, a nie *obok* niej. Jednak przy niewielkim nakładzie pracy można nagiąć menadżer układu do swojej woli. W tym rozdziale „popracujemy nad naszym Swingiem” i oprócz menadżerów układu dowiemy się także czegoś więcej o komponentach. Będziemy je tworzyć, wyświetlać (tam, gdzie *my* będziemy chcieli je umieścić) i wykorzystywać w programie. Nie wygólniej z perspektywy Marysi.

Komponenty biblioteki Swing

Komponent to bardziej poprawne określenie „elementu sterującego”. Czyli tego „czegoś”, co tworzy graficzny interfejs użytkownika aplikacji. *Tego, co użytkownik widzi i z czym prowadzi interakcję.* Pola tekstowe, przyciski, listy przewijane, przyciski opcji itd. — to wszystko są komponenty. W rzeczywistości wszystkie te elementy mają wspólną klasę bazową — javax.swing.JComponent.

Komponenty można zagnieździć

W Swingu niemal we *wszystkich* komponentach można umieszczać inne komponenty. Innymi słowy — *prawie we wszystkim można umieścić cokolwiek innego*. Jednak w większości przypadków będziesz umieszczać *komponenty interakcyjne*, takie jak przyciski bądź listy, w *komponentach tła*, takich jak panele lub ramki. Choć można umieścić, na przykład, panel na przycisku, to jednak byłoby to dosyć dziwaczne i zapewne nie zapewniło zwycięstwa w konkursie na najlepiej zaprojektowany interfejs użytkownika.

Niemniej jednak, za wyjątkiem obiektów JFrame, podział na komponenty *interakcyjne* oraz komponenty *tła* jest sztuczny. Na przykład, JPanel jest zazwyczaj używany jako komponent tła służący do grupowania komponentów, jednak nawet JPanel może być komponentem interakcyjnym. Podobnie jak wszelkie inne komponenty także i JPanel umożliwia rejestrowanie odbiorców generowanych przez niego zdarzeń, takich jak kliknięcia bądź naciśnięcia klawiszy.

Technicznie rzecz biorąc, elementy tworzące graficzny interfejs aplikacji to komponenty biblioteki Swing. Niemal wszystko, co można wyświetlić na ekranie, to obiekty klas potomnych klasy javax.swing.JComponent.

Cztery etapy tworzenia graficznego interfejsu użytkownika (powtórzenie)

- 1 Stworzenie okna (obiekt JFrame):

```
JFrame ramka = new JFrame();
```

- 2 Stworzenie komponentu (przycisku, pola tekstowego lub czegokolwiek innego):

```
JButton przycisk = new JButton("Kliknij mnie");
```

- 3 Dodanie komponentu do ramki:

```
ramka.getContentPane().add(przycisk);
```

- 4 Wyświetlenie ramki (nadanie jej wielkości i określenie, że jest widoczna):

```
ramka.setSize(300,300);  
ramka.setVisible(true);
```

Umieszczaj komponenty interakcyjne:



W komponentach tła:



Menedżery układu

Menedżer układu to obiekt skojarzony z konkretnym komponentem — niemal zawsze z komponentem *tła*. Menedżer zarządza komponentami umieszczonymi *wewnątrz* komponentu, z którym ten menedżer jest skojarzony. Innymi słowy, jeśli ramka zawiera panel, a na panelu został umieszczony przycisk, to menedżer układu panelu określa położenie i wielkość przycisku, natomiast menedżer układu ramki określa położenie i wielkość panelu. Z kolei przycisk nie potrzebuje żadnego menedżera układu, gdyż sam nie zawiera żadnych innych komponentów.

Jeśli panel zawiera pięć komponentów, i nawet jeśli one same posiadają menedżery układu, to i tak wielkość i położenie tych komponentów jest określana przez menedżer układu panelu. Jeśli w tych pięciu komponentach umieszczone są *inne „elementy”*, to zostaną one rozmieszczone przez menedżery układu skojarzone z komponentami, w których „elementy” te są umieszczone.

Mówiąc, że jeden komponent *zawiera* drugi lub że drugi komponent został umieszczony w pierwszym, tak naprawdę mamy na myśli *dodanie* drugiego komponentu do pierwszego; zatem stwierdzenie „panel *zawiera* przycisk”, tak naprawdę oznacza, że przycisk został dodany do panelu przy użyciu, na przykład, takiego wywołania:

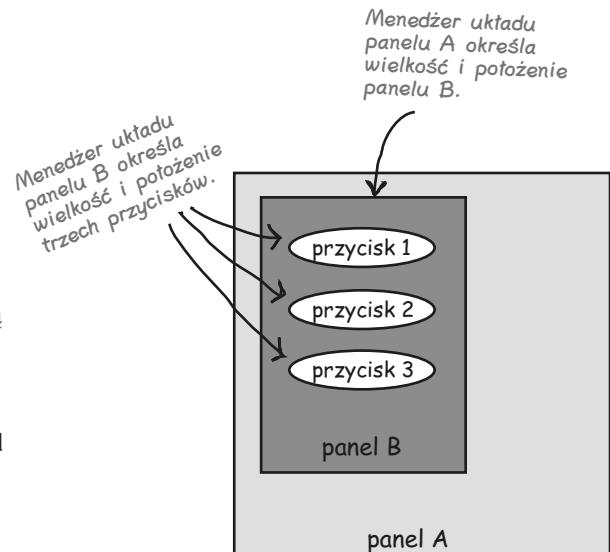
```
panel.add(przycisk);
```

Istnieje kilka rodzajów menedżerów układu, a każdy z komponentów *tła* może mieć swój własny menedżer. Menedżery układu mają własne zasady, według których określają wygląd komponentów. Na przykład, jeden z menedżerów może się starać, by wszystkie komponenty umieszczone w panelu miały tę samą wielkość i zostały umieszczone w prostokątnej siatce, natomiast inny może rozmieszczać komponenty jeden nad drugim. Oto przykład interfejsu użytkownika, w którym jeden panel został zagnieżdżony w innym:

```
JPanel panelA = new JPanel();
JPanel panelB = new JPanel();
panelB.add(new JButton("Przycisk 1"));
panelB.add(new JButton("Przycisk 2"));
panelB.add(new JButton("Przycisk 3"));
panelA.add(panelB);
```



Jako menedżer układu jestem odpowiedzialny za określanie wielkości i położenia komponentów. W tym przykładzie to właśnie ja określiłem, jak duże mają być przyciski i jak powinny być rozmieszczone zarówno względem siebie, jak i względem całej ramki.



Menedżer układu panelu A nie ma ŻADNEGO wpływu na wielkość i położenie trzech przycisków. Hierarchia zarządzania obejmuje wyłącznie jeden poziom — menedżer układu panelu A zarządza jedynie komponentami dodanymi bezpośrednio do tego panelu, nie ma natomiast żadnego wpływu na dalszą zawartość tych komponentów.

W jaki sposób menedżery układu podejmują decyzje?

Różne menedżery układu mają różne zasady określające sposób rozmieszczania komponentów (na przykład mogą je rozmieszczać w siatce, nadawać im takie same wymiary, umieszczać jeden pod drugim i tak dalej), niemniej jednak komponenty mają *niewielki* wpływ na sam proces rozmieszczania. Ogólnie rzecz biorąc, proces rozmieszczania zawartości komponentu tła wygląda mniej więcej tak:

Scenariusz określania układu:

- ① Tworzymy panel i dodajemy do niego trzy przyciski.
- ② Menedżer układu panelu „pyta”, jak duży chce być każdy z przycisków.
- ③ Menedżer układu stosuje swoje zasady rozmieszczania do określenia, czy powinien respektować wszystkie lub tylko niektóre preferencje przycisków, ewentualnie czy w ogóle je zignorować.
- ④ Dodajemy panel do ramki.
- ⑤ Menedżer układu ramki pyta panel, jakie są jego preferowane wymiary.
- ⑥ Menedżer układu ramki wykorzystuje swoje zasady rozmieszczania do określenia, czy powinien respektować wszystkie lub tylko niektóre preferencje panelu, ewentualnie czy w ogóle je zignorować.

Różne menedżery układu mają różne zasady rozmieszczania

Niektoří menedžeri układu respektują preferowane wielkości przycisków. Jeśli przycisk chce mieć wysokość 30 i szerokość 50 pikseli, to właśnie taki obszar menedżer mu przydzieli. Inne menedżery tylko częściowo respektują preferowane wielkości komponentów. Jeśli przycisk chce mieć wysokość 30 i szerokość 50 pikseli, to będzie miał wysokość 30 pikseli, jednak jego szerokość będzie odpowiadać szerokości panelu tła. Jeszcze inne menedżery układu respektują wyłącznie preferencje *największego* z rozmieszczanych komponentów, natomiast wszystkim pozostałym komponentom zostają przypisane takie same wymiary. W niektórych przypadkach operacje wykonywane przez menedżera układu mogą być naprawdę bardzo skomplikowane, jednak w większości przypadków, znając zasady, jakimi dany menedżer się kieruje, można się domyślić, jakie będą wyniki jego działania.

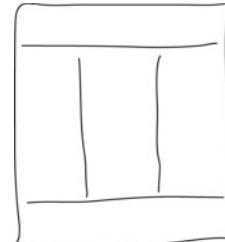


menedżer układu

Wielka trójka menedżerów układu: BorderLayout, FlowLayout oraz BoxLayout

BorderLayout

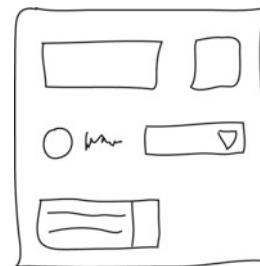
Menedżer BorderLayout dzieli tło komponentu na pięć regionów. W każdym z tych regionów można umieścić tylko jeden komponent. Komponenty rozmieszczane przy wykorzystaniu tego menedżera zazwyczaj nie uzyskują preferowanych wymiarów. BorderLayout **jest domyślnym menedżerem układu stosowanym w ramkach!**



Po jednym komponencie w regionie

FlowLayout

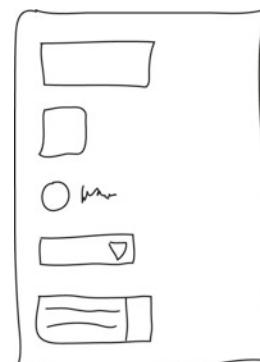
Menedżer FlowLayout działa podobnie do edytorów tekstu, z tą różnicą, iż zamiast słów występują komponenty. Każdy z komponentów ma taką wielkość, jaką chce, a poszczególne komponenty są rozmieszczane od lewej do prawej w kolejności, w jakiej zostały dodane i z wykorzystaniem „opcji przenoszenia do kolejnego wiersza”. Zatem jeśli komponent jest zbyt szeroki, aby umieścić go w dostępnej części „linii”, zostanie przeniesiony do następnej. FlowLayout **jest domyślnym menedżerem układu w panelach!**



Komponenty są dodawane od lewej do prawej i, w razie konieczności, przenoszone do następnej linii.

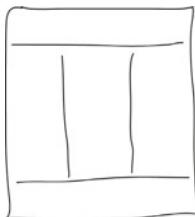
BoxLayout

Pod względem wymiarów nadawanych komponentom menedżer BoxLayout przypomina FlowLayout — w obu tych menedżerach komponenty uzyskują swoje preferowane wymiary. Jednak, w odróżnieniu od FlowLayout, menedżer BoxLayout może rozmieszczać komponenty w pionie (lub w poziomie, ale ta możliwość zazwyczaj nas mniej interesuje). Zatem działa on podobnie jak menedżer FlowLayout, z tym, że zamiast „opcji przenoszenia komponentów do kolejnego wiersza” używany jest specyficzny „klawisz Return”, który wymusza, by każdy komponent został wyświetlony w nowej linii.



Komponenty dodawane od góry do dołu i umieszczone po jednym w „linii”

Menedżer BorderLayout



Menedżer BorderLayout zarządza pięcioma regionami: wschodnim, zachodnim, północnym, południowym i centralnym.

Dodajmy przyciski do regionu wschodniego

```
import javax.swing.*;  
import java.awt.*; ← Klasa BorderLayout należy do pakietu java.awt.  
  
public class Przycisk1 {  
    public static void main(String[] args) {  
        Przycisk1 gui = new Przycisk1();  
        gui.doRoboty();  
    }  
  
    public void doRoboty() {  
        JFrame ramka = new JFrame();  
        JButton przycisk = new JButton("Kliknij mnie"); ← Określ region  
        ramka.getContentPane().add(BorderLayout.EAST, przycisk);  
        ramka.setSize(200,200);  
        ramka.setVisible(true);  
    }  
}
```



**WYTĘŻ
UMYSŁ**

W jaki sposób menedżer BorderLayout wyznaczył taką wielkość przycisku?

Jakie są czynniki, które musi uwzględnić?

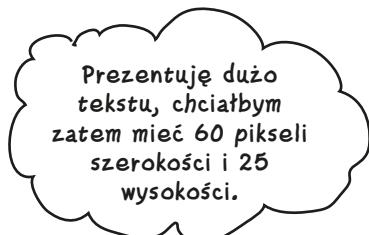
Dlaczego przycisk nie jest szerszy lub wyższy?



Zobacz, co się stanie, jeśli na przycisku wyświetlimy więcej znaków...

```
public void doRoboty() {
    JFrame ramka = new JFrame();
    JButton przycisk = new JButton("Kliknij mnie jeśli chcesz!");
    ramka.getContentPane().add(BorderLayout.EAST, przycisk);
    ramka.setSize(200,200);
    ramka.setVisible(true);
}
```

Zmieniliśmy jedynie tekst wyświetlany na przycisku.



Ponieważ przycisk został umieszczony we wschodnim regionie układu, zatem zgodzę się na jego preferowaną szerokość, ale nie obchodzi mnie, jak wysoki chciałbym być przycisk – ze względu na moje zasady, będzie miał taką samą wysokość jak cała ramka.



Następny razem skorzystam z usług menedżera FlowLayout – on mi zapewni wszystko, o co poproszę.



Przycisk uzyskuje preferowaną szerokość, jednak wysokość jest określana przez menedżer uktadu.

Menedżer BorderLayout

Spróbujmy umieścić przycisk w regionie północnym

```
public void doRoboty() {  
    JFrame ramka = new JFrame();  
    JButton przycisk = new JButton("Nie ma, nie ma wody na...");  
    ramka.getContentPane().add(BorderLayout.NORTH, przycisk);  
    ramka.setSize(200,200);  
    ramka.setVisible(true);  
}
```



Przycisk ma preferowaną wysokość, jednak jego szerokość odpowiada szerokości ramki.

A teraz, niech przycisk poprosi, by nadać mu większą wysokość

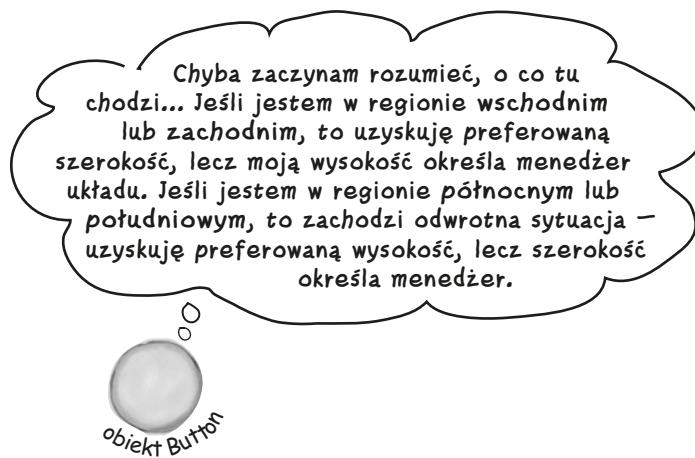
W jaki sposób można to osiągnąć? Przycisk już jest tak szeroki, jak to możliwe — jego szerokość odpowiada szerokości ramki. Jednak możemy sprawić, by był wyższy, wyświetlając jego opis większą czcionką.

```
public void doRoboty() {  
    JFrame ramka = new JFrame();  
    JButton przycisk = new JButton("Kliknij TU!");  
    Font duzaCzcionka = new Font("serif", Font.BOLD, 28);  
    przycisk.setFont(duzaCzcionka);  
    ramka.getContentPane().add(BorderLayout.NORTH, przycisk);  
    ramka.setSize(200,200);  
    ramka.setVisible(true);  
}
```



Większa czcionka zmusi ramkę do przydzielenia przyciskowi obszaru o większej wysokości.

Choć szerokość przycisku się nie zmieniła, to jednak obecnie jest on wyższy. Północny region układu został powiększony i dostosowany do nowej preferowanej wysokości przycisku.



Co się jednak dzieje
w regionie centralnym?

Region centralny zajmuje pozostały obszar!

(z jednym wyjątkiem, którym zaraz się zajmiemy).

```
public void doRoboty() {
    JFrame ramka = new JFrame();

    JButton przyciskW = new JButton("Wschód");
    JButton przyciskZ = new JButton("Zachód");
    JButton przyciskPn = new JButton("Północ");
    JButton przyciskPd = new JButton("Południe");
    JButton przyciskC = new JButton("Centrum");

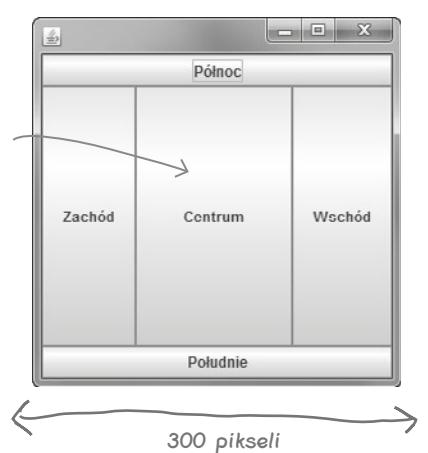
    ramka.getContentPane().add(BorderLayout.EAST , przyciskW);
    ramka.getContentPane().add(BorderLayout.WEST , przyciskZ);
    ramka.getContentPane().add(BorderLayout.NORTH , przyciskPn);
    ramka.getContentPane().add(BorderLayout.SOUTH , przyciskPd);
    ramka.getContentPane().add(BorderLayout.CENTER , przyciskC);

    ramka.setSize(300,300);
    ramka.setVisible(true);
}
```

Wymiary komponentu umieszczonego w regionie centralnym będą odpowiadać wielkości obszaru pozostałego w ramce po rozmieszczeniu komponentów we wszystkich pozostałych regionach. (W tym przykładzie ramka ma wymiary 300x300 pikseli).

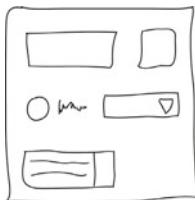
Komponenty w regionach wschodnim i zachodnim uzyskują preferowaną szerokość.

Komponenty w regionach północnym i południowym uzyskują preferowaną wysokość.



Jeśli umieścisz jakieś komponenty w regionie północnym lub południowym, to ich szerokość będzie odpowiadać szerokości tej ramki, zatem komponenty umieszczone w regionie wschodnim i zachodnim nie będą tak wysokie, jak mogłyby być, gdyby regiony północny i południowy pozostały puste.

Menedżer FlowLayout



Menedżer FlowLayout zwraca uwagę na ciągłość ułożenia komponentów — są one rozmieszczane do lewej do prawej oraz od góry ku dołowi w kolejności dodawania.

Dodajmy panel do regionu wschodniego:

Domyślnym menedżerem układu komponentów JPanel jest menedżer FlowLayout. W przypadku umieszczenia panelu w ramce wielość oraz rozmieszczenie samego panelu wciąż są określane przez menedżer BorderLayout, jednak wszelkie komponenty umieszczone wewnątrz panelu (innymi słowy komponenty dodane do panelu przy użyciu wywołania panel.add(komponent)) znajdują się pod kontrolą menedżera FlowLayout. Zaczniemy od umieszczenia w ramce pustego panelu, a na następnej stronie dodamy do tego panelu komponenty...

```
import javax.swing.*;
import java.awt.*;

public class Panel1 {
    public static void main(String[] args) {
        Panel1 gui = new Panel1();
        gui.doRoboty();
    }

    public void doRoboty() {
        JFrame ramka = new JFrame();
        JPanel panel = new JPanel();
        panel.setBackground(Color.darkGray);
        ramka.getContentPane().add(BorderLayout.EAST, panel);
        ramka.setSize(200,200);
        ramka.setVisible(true);
    }
}
```

Panel jest pusty, zatem preferowana szerokość, o jaką poprosit, nie jest zbyt duża.



Panel będzie szary, aby można łatwo określić, w którym miejscu ramki się znajduje.

Dodajmy przycisk do panelu

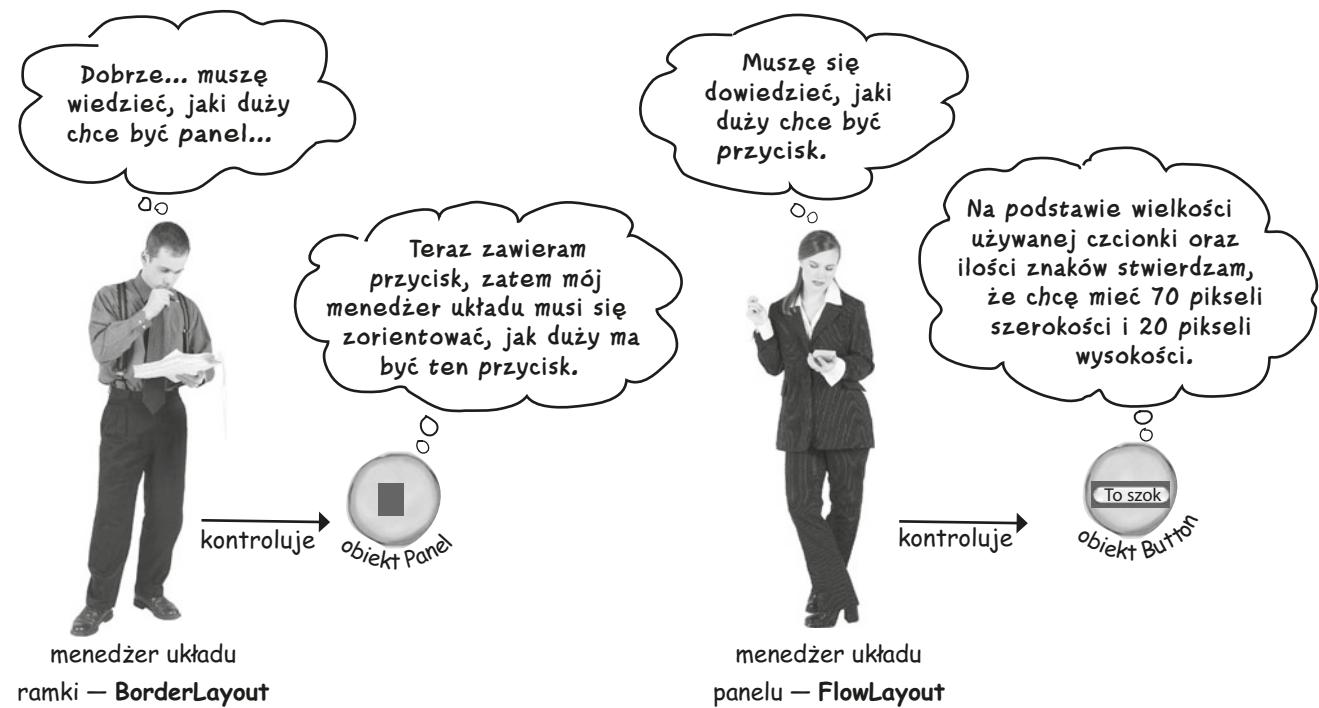
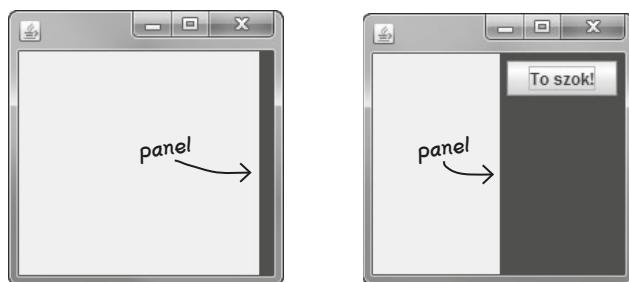
```
public void doRoboty() {
    JFrame ramka = new JFrame();
    JPanel panel = new JPanel();
    panel.setBackground(Color.darkGray);

    JButton przycisk = new JButton("To szok!");

    panel.add(przycisk);
    ramka.getContentPane().add(BorderLayout.EAST, panel);

    ramka.setSize(200,200);
    ramka.setVisible(true);
}
```

Dodajemy przycisk do panelu, a panel do ramki. Menedżer układu panelu (FlowLayout) określa wielkość przycisku, a menedżer ramki (BorderLayout) – wielkość panelu.



Menedżer FlowLayout

Co się stanie, jeśli do panelu dodamy DWA przyciski?

```
public void doRoboty() {  
    JFrame ramka = new JFrame();  
    JPanel panel = new JPanel();  
    panel.setBackground(Color.darkGray);  
  
    JButton przycisk = new JButton("To szok!"); ← Tworzymy dwa przyciski.  
    JButton przycisk2 = new JButton("Trach!"); ←  
  
    panel.add(przycisk); ← Dodajemy OBA przyciski do panelu!  
    panel.add(przycisk2); ←  
  
    ramka.getContentPane().add(BorderLayout.EAST, panel);  
    ramka.setSize(200,200);  
    ramka.setVisible(true);  
}
```

To co chcieliśmy uzyskać:



Chcieliśmy, aby przyciski były wyświetlane jeden nad drugim.

To co uzyskaliśmy:



Panel został rozszerzony w taki sposób, aby oba przyciski mogły być wyświetlane jeden obok drugiego.

Zauważ, że przycisk "Trach!" jest mniejszy od przycisku "To szok!"... właśnie w taki sposób działa menedżer FlowLayout. Przycisk uzyskuje tylko takie wymiary, jakie musi mieć (i nic ponad to).



Zaostrz ołówek

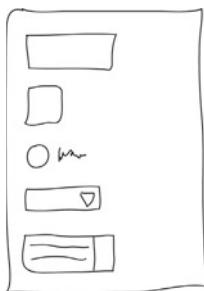
W jaki sposób zmieniłby się interfejs użytkownika, gdyby powyższy kod zmodyfikować w następujący sposób.

```
JButton przycisk = new JButton("To szok!");  
JButton przycisk2 = new JButton("Trach!");  
JButton przycisk3 = new JButton("Och?!");  
panel.add(przycisk);  
panel.add(przycisk2);  
panel.add(przycisk3);
```



Narysuj jak, według Ciebie, wyglądałby interfejs użytkownika po uruchomieniu takiego programu.

(A następnie go uruchom!)



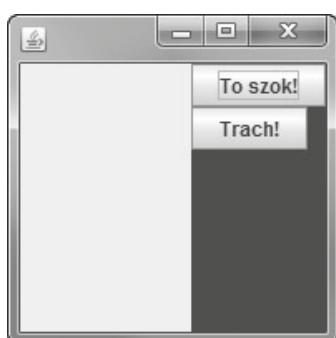
Menedżer BoxLayout spieszy z pomocą!

Ten menedżer rozmieszcza komponenty jeden nad drugim, nawet jeśli można by je umieścić obok siebie.

W odróżnieniu od menedżera FlowLayout, menedżer BoxLayout może wymusić wyświetlenie komponentu w nowym wierszu, nawet jeśli jest na tyle miejsca, że oba komponenty można by wyświetlić obok siebie.

Jednak w takim razie trzeba będzie zmienić menedżer układu panelu z domyślnego FlowLayout na BoxLayout.

```
public void doRoboty() {
    JFrame ramka = new JFrame();
    JPanel panel = new JPanel();
    Zmieniamy menedżera układu — aktualnie
    będzie nim nowy obiekt klasy BoxLayout.
    panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
    Konstruktor obiektu BoxLayout musi wiedzieć,
    na jakim komponencie menedżer ma operować
    (w tym przypadku jest to panel) oraz wzduż jakiej osi
    należy rozmieszczać komponenty (aby komponenty były
    umieszczane jeden nad drugim, musimy je rozmieszczać
    wzduż osi Y — stąd używamy statej Y_AXIS).
    panel.setBackground(Color.darkGray);
    JButton przycisk = new JButton("To szok!");
    JButton przycisk2 = new JButton("Trach!");
    panel.add(przycisk);
    panel.add(przycisk2);
    ramka.getContentPane().add(BorderLayout.EAST, panel);
    ramka.setSize(200,200);
    ramka.setVisible(true);
}
```



Zwrć uwagę, że panel stał się węższy, gdyż oba przyciski nie muszą już być umieszczane jeden obok drugiego. A zatem panel poinformował, że potrzebuje tylko tyle miejsca, aby zmieścić się w nim najszerzszy przycisk — przycisk „To szok!”.

Nie istnieja
główne pytania

P: Dlaczego komponentów nie można dodawać do ramki bezpośrednio, w taki sam sposób jak do panelu?

O: Komponent JFrame jest szczególny, gdyż to właśnie on pośredniczy w faktycznym wyświetlaniu komponentów na ekranie. Wszystkie komponenty Swing są w całości napisane w Javie, jednak JFrame musi dysponować jakimś połączeniem z systemem operacyjnym, aby mieć możliwość wyświetlania swej zawartości. Panel zawartości można sobie wyobrażać jako warstwę napisaną w całości w Javie, która jest umieszczona powyżej komponentu JFrame. Ewentualnie można sobie wyobrazić, że JFrame jest ramką okna, a panel zawartości szybą. W końcu... okna mają szyby. Można nawet zamienić domyślny panel okna na swój własny obiekt JPanel, aby to Twój obiekt był stosowany jako panel zawartości okna. W tym celu należy użyć wywołania:

```
rama.setContentPane(mojPanel);
```

P: Czy można zmienić menedżer układu ramki?
Co zrobić, jeśli będę chciał, aby ramka używała menedżera FlowLayout, a nie BorderLayout?

O: Najprostszym sposobem uzyskania takiego efektu jest stworzenie panelu, skonstruowanie interfejsu użytkownika w oparciu o ten panel, a następnie zastąpienie nim domyślnego panelu zawartości ramki (przy użyciu kodu przedstawionego w poprzedniej odpowiedzi).

P: A co zrobić, gdybym chciał określić inne preferowane wymiary komponentu? Czy jest jakaś metoda setSize()?

O: Tak, jest metoda setSize(), jednak menedżery układu ignorują ją. Istnieje bowiem różnica pomiędzy preferowaną wielkością komponentu a wielkością, jaką chce mu nadać programista. Wielkość preferowaną określa się na podstawie obszaru, jakiego komponent faktycznie potrzebuje (i tę decyzję komponent podejmuje samodzielnie). Menedżer układu wywołuje metodę getPreferredSize() komponentu, a ta metoda nie zwraca najmniejszej uwagi na to, czy wcześniej wywołano metodę setSize().

P: A czy nie mogę po prostu wyświetlać komponentów tam, gdzie bym tego chciał? Czy nie można wyłączyć menedżerów układu?

O: Można. W każdym z tworzonych komponentów można wywoływać metodę setLayout(null), co sprawi, że to w Twojej gestii będzie precyzyjne określenie położenia komponentów oraz ich wymiarów. Jednak na dłuższą metę niemal zawsze wygodniejsze jest stosowanie menedżerów układu.



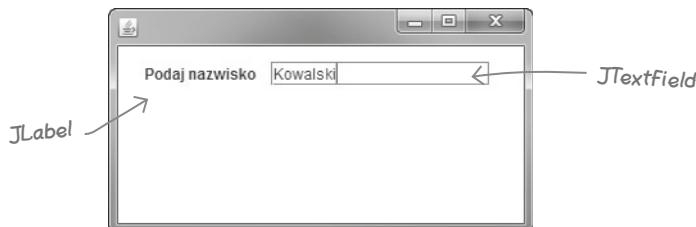
CELNE SPOSTRZEŻENIA

- Menedżery układu określają wymiary i położenie komponentów umieszczonych w innych komponentach.
- Dodając komponent do innego komponentu (czasami określonego jako komponent *ta*, choć rozróżnienie to nie wynika z przyczyn technicznych), wymiary i położenie komponentu dodawanego określone są przez menedżer układu komponentu *ta*.
- Przed podjęciem decyzji dotyczących postaci układu menedżer „pyta” komponenty o ich preferowane wymiary. W zależności od zasad stosowanych przez konkretny menedżer, może on wszystkie lub tylko niektóre z preferencji komponentów zignorować.
- Menedżer BorderLayout pozwala na umieszczanie komponentów w pięciu regionach. Dodając komponent, należy określić, w którym z regionów go umieścić; w tym celu używane jest następujące wywołanie:
`add(BorderLayout.EAST, panel);`
- W przypadku użycia menedżera BorderLayout komponenty umieszczone w regionie północnym i południowym mają preferowaną wysokość, lecz ich szerokość określa menedżer. Z kolei komponenty umieszczone w regionach wschodnim i zachodnim mają preferowaną szerokość, a ich wysokość określa menedżer. Komponent umieszczony w regionie centralnym zajmuje cały pozostały obszar (chyba że wywołana zostanie metoda `pack()`).
- Metoda `pack()` jest używana do optymalnego „upakowania” komponentów — komponent centralny uzyskuje pełne preferowane wymiary, a następnie metoda określa wymiary całej ramki, dostosowując je do wymiarów komponentów umieszczonych w pozostałych regionach.
- Menedżer FlowLayout rozmieszcza komponenty od lewej do prawej i od góry do dołu, w kolejności, w jakiej były one dodawane; przy czym, komponenty są przenoszone do kolejnego „wiersza” tylko wtedy, gdy nie mieścią się w poprzednim.
- W przypadku użycia menedżera FlowLayout komponenty uzyskują oba preferowane wymiary.
- Menedżer BoxLayout pozwala na wyświetlanie komponentów jeden nad drugim, nawet jeśli mogłyby się zmieścić obok siebie. Także ten menedżer respektuje preferowane wymiary komponentów.
- BorderLayout jest domyślnym menedżerem układu stosowanym w ramkach; z kolei, domyślnym menedżerem w panelach jest FlowLayout.
- Jeśli chcesz zmienić menedżer układu stosowany w panelu, wywołaj metodę `setLayout()` tego panelu

Zabawy z komponentami biblioteki Swing

Poznałeś już podstawowe informacje dotyczące menedżerów układu, nadszedł zatem czas, aby wypróbować kilka spośród najczęściej używanych komponentów, takich jak pola tekstowe, przewijane wielowierszowe pola tekstowe, pola wyboru czy też listy. Nie będziemy przedstawiać wszystkich metod każdego z tych komponentów — opiszymy jedynie wybrane z nich, aby ułatwić Ci naukę stosowania danego komponentu.

JTextField



Konstruktory

```
JTextField pole = new JTextField(20);
JTextField pole = new JTextField("Podaj nazwisko");
```

Liczba 20 oznacza 20 kolumn, a nie 20 pikseli. To właśnie na podstawie tej wartości określana jest preferowana szerokość pola tekstowego.

Jak go używać?

- 1 Pobieranie tekstu z pola:

```
System.out.println(pole.getText());
```

- 2 Zapisywanie tekstu w polu:

```
pole.setText("cokolwiek");
pole.setText("");
```

To wywołanie usuwa całą zawartość pola tekstowego.

- 3 Odbieranie zdarzeń ActionEvent, gdy użytkownik naciśnie w polu klawisz Return lub Enter:

```
pole.addActionListener(mojActionListener);
```

Można także zarejestrować odbiorce, do którego będą przekazywane zdarzenia związane z naciśnięciem klawiszy, oczywiście o ile naprawdę chcesz wiedzieć o każdym klawisz, który został naciśnięty przez użytkownika.

- 4 Zaznaczenie tekstu w polu:

```
pole.selectAll();
```

- 5 Umieszczenie kurSORA w polu (dzięki czemu użytkownik może zacząć wpisywać w nim tekst):

```
pole.requestFocus();
```

JTextArea



W odróżnieniu od pól tekstowych JTextField w komponentach JTextArea można wpisywać więcej niż jeden wiersz tekstu. Komponenty te trzeba jednak skonfigurować, gdyż domyślnie nie są wyposażane w paski przewijania ani nie jest włączana opcja przenoszenia wyrazów do kolejnego wiersza. Aby umożliwić przewijanie zawartości komponentu JTextArea, należy umieścić go w komponencie JScrollPane. JScrollPane to komponent, który wprost uwielbia przewijać i z przyjemnością zaspokoi wszelkie potrzeby JTextArea, oczywiście te związane z przewijaniem zawartości.

Konstruktor

```
JTextArea tekst = new JTextArea(10, 20);
```

Liczba 10 oznacza 10 wierszy
(i ma wpływ na preferowaną wysokość).
Liczba 20 oznacza 20 kolumn
(i ma wpływ na preferowaną szerokość).

Jak go używać?

1 Stworzenie pola zawierającego wyłącznie pionowy pasek przewijania:

```
JScrollPane przewijanie = new JScrollPane(tekst);  
tekst.setLineWrap(true);  
przewijanie.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);  
przewijanie.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);  
panel.add(przewijanie);
```

Tworzymy komponent JScrollPane, i przekazujemy do niego pole tekstowe, którego zawartość ma być przewijana.

Te dwa wywołania informują komponent JScrollPane, że ma wyświetlać wyłącznie pionowy pasek przewijania.

To ważne!!! Najpierw należy przekazać pole tekstowe do komponentu JScrollPane (w wywołaniu jego konstruktora), a następnie dodać ten komponent do panelu. Pamiętaj, że pola tekstowego nie należy dodawać bezpośrednio do panelu!

2 Zastąpienie tekstu umieszczonego w polu:

```
tekst.setText("Nie wszyscy zagubieni błądzą.");
```

3 Dodanie tekstu do bieżącej zawartości pola:

```
tekst.append("przycisk został kliknięty");
```

4 Zaznaczenie całego tekstu wpisanego w polu:

```
tekst.selectAll();
```

5 Umieszczenie kurSORA z powrotem w polu (dzięki czemu użytkownik może wpisywać w nim tekst):

```
tekst.requestFocus();
```

Przykład zastosowania komponentu JTextArea

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class PoleTekstoweWW1 implements ActionListener {

    JTextArea tekst;

    public static void main(String[] args) {
        PoleTekstoweWW1 gui = new PoleTekstoweWW1();
        gui.tworzGUI();
    }

    public void tworzGUI() {
        JFrame ramka = new JFrame();
        JPanel panel = new JPanel();
        ramka.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JButton przycisk = new JButton("Po prostu kliknij");
        przycisk.addActionListener(this);
        tekst = new JTextArea(10,20);
        tekst.setLineWrap(true);

        JScrollPane przewijanie = new JScrollPane(tekst);
        przewijanie.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        przewijanie.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        panel.add(przewijanie);

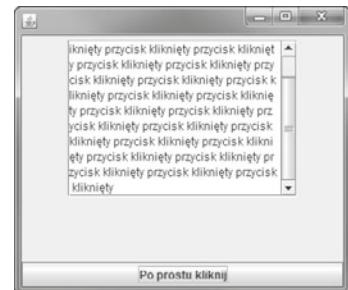
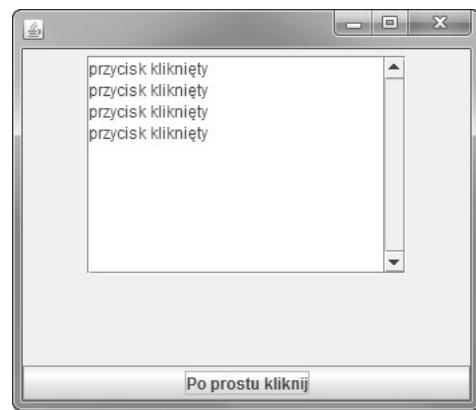
        ramka.getContentPane().add(BorderLayout.CENTER, panel);
        ramka.getContentPane().add(BorderLayout.SOUTH, przycisk);

        ramka.setSize(350,300);
        ramka.setVisible(true);
    }
}

public void actionPerformed(ActionEvent zdarzenie) {
    tekst.append("przycisk kliknięty \n");
}

```

Dodaj znak nowego wiersza, aby za każdym razem, gdy zostanie kliknięty przycisk, tańcuch "przycisk kliknięty" był dodawany w nowym wierszu. W przeciwnym razie nowe tańcuchy będą dodawane w tym samym wierszu, w którym został umieszczony poprzedni.



JCheckBox



Nie istnieja głupie pytania

P: Czy te całe menedżery układu nie przysparzają więcej problemów niż pozytku? Jeśli muszę narażać się na te wszystkie trudności, to również dobrze mógłbym na stałe podać współrzędne położenia i wymiary komponentów.

O: Uzyskanie dokładnie takiego wyglądu interfejsu użytkownika, jaki byśmy sobie życzyli przy wykorzystaniu menedżera układu może być trudnym wyzwaniem. Jednak pomyśl o tym, co tak naprawdę menedżer układu dla Ciebie robi. Nawet pozornie proste zadanie określenia, w jakich miejscach powinny się znaleźć poszczególne komponenty, może być trudne. Na przykład, menedżer układu zapobiega sytuacjom, w których komponenty wzajemnie na siebie zachodzą. Innymi słowy, menedżer wie, w jaki sposób określać odstęp pomiędzy elementami (oraz pomiędzy krawędziami ramki). Oczywiście, że możesz to wszystko zrobić sam. Ale wyobraź sobie, co się stanie, jeśli będziesz chciał wyświetlić komponenty możliwie blisko siebie? Możesz je ręcznie rozmieścić tuż obok siebie, jednak ich wygląd może być poprawny tylko na Twojej wirtualnej maszynie Javy.

Dlaczego? Ponieważ komponenty mogą wyglądać nieco inaczej na poszczególnych platformach systemowych, zwłaszcza jeśli wykorzystują „rodzimy” wygląd elementów sterujących charakterystyczny dla danej platformy. Czynniki takie jak postać wierzchołków komponentów mogą sprawić, że śliczne komponenty, które na jednej platformie były do siebie perfekcyjnie dopasowane, na drugiej będą się wzajemnie przesłaniać.

A wciąż jeszcze nie doszliśmy do najważniejszego aspektu działania menedżerów układu. Pomyśl, co się stanie, jeśli użytkownik zmieni wymiary okna! Albo jeśli graficzny interfejs aplikacji zmienia się dynamicznie — czyli jego elementy znikają i pojawiają się. Jeśli sam musiałby zajmować się rozmieszczaniem komponentów po każdej zmianie wielkości okna lub zawartości komponentu tła... o rany.

Konstruktor

```
JCheckBox poleWyboru = new JCheckBox("Grasz dalej?");
```

Jak go używać?

- 1 Odbieranie zdarzeń generowanych przez pole wyboru (kiedy użytkownik je zaznaczy lub usunie z niego znacznik):

```
poleWyboru.addItemListener(this);
```

- 2 Obsługa zdarzenia (i określenie, czy pole jest zaznaczone czy nie):

```
public void itemStateChanged(ItemEvent zdarzenie) {  
    String wlaczoneCzyNie = "wyłączone";  
    if (poleWyboru.isSelected()) wlaczoneCzyNie = "włączone";  
    System.out.println("Pole jest " + wlaczoneCzyNie);  
}
```

- 3 Programowe zaznaczenie pola lub usunięcie jego zaznaczenia:

```
poleWyboru.setSelected(true);  
poleWyboru.setSelected(false);
```

JList



Konstruktor komponentu `JList` oczekuje podania tablicy typu `Object`. Nie muszą to byćłańcuchy znaków (obiekty `String`), jednak na liście będą wyświetlane właśniełańcuchy znaków.

Konstruktor

```
String[] listaOpcji = {"alfa", "beta", "gama", "delta",
                      "epsilon", "zeta", "eta", "teta"};  
  
JList lista = new JList(listaOpcji);
```

Jak go używać?

- 1 Dodanie pionowego paska przewijania do listy:

```
JScrollPane przewijanie = new JScrollPane(lista);
przewijanie.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
przewijanie.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(przewijanie);
```

Zasada jest taka sama jak w przypadku komponentów `JTextArea` — należy stworzyć komponent `JScrollPane` (i przekazać do niego listę), a następnie dodać ten komponent (a NIE listę) do panelu.

- 2 Określenie ilości wierszy widocznych na liście:

```
lista.setVisibleRowCount(6);
```

- 3 Ograniczenie możliwości zaznaczania tylko do jednego elementu naraz:

```
lista.setSelectionMode(ListSelectionMode.SINGLE_SELECTION);
```

- 4 Rejestracja odbiorcy zdarzeń związanych z wybieraniem opcji listy:

```
lista.addListSelectionListener(this);
```

- 5 Obsługa zdarzeń (i określenie, która opcja listy została zaznaczona):

```
public void valueChanged(ListSelectionEvent zdarzenie) { ←
    if (!zdarzenie.getValueIsAdjusting()) {
        String opcja = (String) lista.getSelectedValue();
        System.out.println(opcja);
    }
}
```

Jeśli nie umieścisz kodu obsługującego zdarzenie w instrukcji if, zostanie on wykonany DWA razy.

Metoda `getSelectedValue()` zwraca obiekt klasy `Object`. Na liście nie muszą być umieszczane jedynie obiekty `String`.



Ta część rozdziału jest opcjonalna. Stworzymy w niej pełną wersję aplikacji MuzMachina, wyposażoną w graficzny interfejs użytkownika i wszelkie inne opcje. W kolejnym rozdziale — „Zapisywanie obiektów” — nauczysz się zapisywać i odtwarzać kompozycje. W końcu, w rozdziale poświęconym zagadnieniom sieciowym (zatytułowanym „Nawiąż połączenie”) zmienimy naszą aplikację w klienta pogawędek sieciowych.

Tworzenie aplikacji MuzMachina

W dalszej części tego rozdziału przedstawiona zostanie wersja programu MuzMachina wyposażona w przyciski umożliwiające rozpoczęcie i zatrzymywanie odtwarzania oraz zmianę tempa. Zamieszczony kod źródłowy jest kompletny i dokładnie opisany, jednak poniżej przedstawiliśmy dodatkowy opis aplikacji:

- ① Stworzymy graficzny interfejs użytkownika składający się z 256 przycisków opcji (JCheckBox, które początkowo nie będą zaznaczone), 16 etykiet (JLabel) określających nazwy instrumentów oraz czterech przycisków.
- ② W każdym z czterech przycisków zarejestrujemy odbiorcę zdarzeń ActionListener. Nie musimy tworzyć odbiorców dla poszczególnych pól wyboru, gdyż nie chcemy dynamicznie zmieniać odtwarzanych dźwięków (czyli wprowadzać zmian w odtwarzanej muzyce w odpowiedzi na zaznaczanie lub usuwanie zaznaczenia pola wyboru). Zamiast tego będziemy czekać, aż użytkownik kliknie przycisk „Start”, a następnie sprawdzimy wszystkie 256 pól wyboru, określmy ich stan i na jego podstawie stworzymy ścieżkę MIDI.
- ③ Przygotujemy system MIDI do użycia (robiliśmy to już wcześniej), czyli pobierzemy obiekt sekwensera (Sequencer), a następnie stworzymy sekwencję (Sequence) oraz ścieżkę (Track). Wykorzystamy metodę sekwensera setLoopCount(), która została dodana dopiero w wersji 5.0 Javy. Pozwala ona określić, ile razy sekwencja ma zostać odtworzona. Skorzystamy także z możliwości określenia współczynnika tempa sekwencji, który pozwoli nam zmieniać szybkość jej odtwarzania i zachowywać ją pomiędzy kolejnymi odtworzeniami sekwencji.
- ④ Prawdziwa praca zacznie się w chwili, gdy użytkownik kliknie przycisk „Start”. Metoda obsługująca zdarzenia generowane przez ten przycisk wywoła metodę utworzSciezkeIOdtworz(). W tej metodzie przejrzymy 256 pól wyboru (wiersz po wierszu, analizując kolejnych 16 taktów dla każdego instrumentu) i określmy ich stan, a następnie wykorzystamy te informacje do stworzenia ścieżki MIDI (przy tym skorzystamy z pomocniczej metody tworzZdarzenie() przedstawionej w poprzednim rozdziale). Kiedy ścieżka będzie gotowa, uruchomimy sekwenser, który będzie w kółko odtwarzać naszą kompozycję aż do momentu, gdy użytkownik kliknie przycisk „Stop”.

Kod aplikacji MuzMachina

```
import java.awt.*;
import javax.swing.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;

public class MuzMachina implements MetaEventListener {
    JPanel panelGlowny;
    ArrayList<JCheckBox> listaPolWyboru; ← Wszystkie pola wyboru umieszczamy
    Sequencer sekwenser;
    Sequence sekwencja;
    Track sciezka;
    JFrame ramkaGlowna;

    String[] nazwyInstrumentow = {"Bass Drum", "Closed Hi-Hat", "Open Hi-Hat", "Acoustic Snare",
                                   "Crash Cymbal", "Hand Clap", "High Tom", "Hi Bongo", "Maracas",
                                   "Whistle", "Low Conga", "Cowbell", "Vibraslap", "Low-mid Tom",
                                   "High Agogo", "Open Hi Conga"};

    int[] instrumenty = {35, 42, 46, 38, 49, 39, 50, 60, 70, 72, 64, 56, 58, 47, 67, 63};

    public static void main (String[] args) {
        new MuzMachina().tworzGUI();
    }

    public void tworzGUI() {
        ramkaGlowna = new JFrame("MuzMachina");
        ramkaGlowna.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        BorderLayout uklad = new BorderLayout();
        JPanel panelTla = new JPanel(uklad);
        panelTla.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        listaPolWyboru = new ArrayList<JCheckBox>();
        Box obszarPrzyciskow = new Box(BoxLayout.Y_AXIS);

        JButton start = new JButton("Start");
        start.addActionListener(new MojStartListener());
        obszarPrzyciskow.add(start);

        JButton stop = new JButton("Stop");
        stop.addActionListener(new MojStopListener());
        obszarPrzyciskow.add(stop);

        JButton tempoG = new JButton("Szybciej");
        tempoG.addActionListener(new MojTempoGListener());
        obszarPrzyciskow.add(tempoG);

        JButton tempoD = new JButton("Wolniej");
        ← Oto nazwy instrumentów podane jako ciągi znaków; będą one używane przy tworzeniu graficznego interfejsu użytkownika, gdzie każda z nich zostanie wyświetlona w osobnym wierszu.
        ← Te liczby reprezentują faktyczne nuty grane na bębnie. Można je porównać z klawiszami fortepianu. Kanat bębna przypomina kanat fortepianu, jedyna różnica pomiędzy nimi polega na tym, iż „klawisze” pianina odpowiadają w tym przypadku innym bębnom. I tak, klawisz numer 35 to Bass drum, a klawisz 42 to Closed Hi-Hat i tak dalej.
        ← Tak zwana „pusta ramka” pozwala na stworzenie odstępów pomiędzy krawędziami panelu a obszarem, w którym zostały umieszczone komponenty. Pełni ona funkcję czysto estetyczną.
    }
}

← Ten fragment nie zawiera niczego szczególnego, po prostu trochę kodu tworzącego graficzny interfejs użytkownika. Jego znacząną część widziałeś już wcześniej.
```

```

tempoD.addActionListener(new MojTempoDListener());
obszarPrzyciskow.add(tempoD);

Box obszarNazw = new Box(BoxLayout.Y_AXIS);
for (int i = 0; i < 16; i++) {
    obszarNazw.add(new Label(nazwyInstrumentow[i]));
}

panelTla.add(BorderLayout.EAST, obszarPrzyciskow);
panelTla.add(BorderLayout.WEST, obszarNazw);

ramkaGlowna.getContentPane().add(panelTla);

GridLayout siatkaPolWyboru = new GridLayout(16,16);
siatkaPolWyboru.setVgap(1);
siatkaPolWyboru.setHgap(2);
panelGlowny = new JPanel(siatkaPolWyboru);
panelTla.add(BorderLayout.CENTER, panelGlowny);

for (int i = 0; i < 256; i++) {
    JCheckBox c = new JCheckBox();
    c.setSelected(false);
    listaPolWyboru.add(c);
    panelGlowny.add(c);
} // koniec pętli

konfigurujMidi();

ramkaGlowna.setBounds(50,50,300,300);
ramkaGlowna.pack();
ramkaGlowna.setVisible(true);
} // koniec metody

public void konfigurujMidi() {
    try {
        sekvenser = MidiSystem.getSequencer();
        sekvenser.open();
        sekwencja = new Sequence(Sequence.PPQ,4);
        sciezka = sekwencja.createTrack();
        sekvenser.setTempoInBPM(120);
    } catch(Exception e) {e.printStackTrace();}
} // koniec metody

```

To wciąż kod tworzący graficzny interfejs użytkownika.
Nic godnego uwagi.

Tworzymy pola wyboru, przypisujemy każdemu z nich wartość false (przez co nie są zaznaczone) i dodajemy do obiektu ArrayList ORAZ do panelu.

Standardowy kod konfiguracji MIDI służący do pobrania obiektów Sequencer, Sequence oraz Track. Wciąż nic, na co warto by zwrócić uwagę.

Kod aplikacji MuzMachine

W tej metodzie wszystko się dzieje! To właśnie w niej zamieniamy stany poszczególnych przycisków opcji na zdarzenia MIDI, a te dodajemy do ścieżki.

```
public void utworzSciezkeIOdtworz() {  
    int[] listaSciezki = null; ← Stworzymy 16-elementową tablicę, która postuży nam do przechowywania informacji o wszystkich 16 takach dla konkretnego instrumentu. Jeśli instrument ma grać w danym taktie, to wartością odpowiedniego elementu tablicy będzie numer klawisza. Z kolei, jeśli w danym taktie instrument nie ma grać, to w odpowiednim elemencie tablicy zostanie umieszczona wartość 0.  
  
    sekwencja.deleteTrack(sciezka); } ← Usuwamy starą ścieżkę i tworzymy nową.  
  
    sciezka = sekwencja.createTrack(); ← Kolejne czynności wykonujemy dla WSZYSTKICH szesnastu wierszy (to znaczy: Bass, Congo itd.).  
  
    for (int i = 0; i < 16; i++) { ← Zapisujemy wartość „klucza”; określa on, jaki instrument jest używany (Bass, Hi-Hat itd.; tablica instrumenty przechowuje faktyczne numery MIDI każdego z instrumentów).  
        listaSciezki = new int[16];  
  
        int klucz = instrumenty[i]; ← Kolejne czynności wykonujemy dla każdego z szesnastu TAKTÓW w wierszu.  
  
        for (int j = 0; j < 16; j++ ) { ← Czy pole wyboru w danym taktie jest zaznaczone? Jeśli tak, to w tablicy (w komórce reprezentującej dany takt) zapisujemy wartość klucza. Jeśli pole wyboru nie jest zaznaczone, oznacza to, że instrument nie ma grać w tym taktie, zatem w odpowiedniej komórce tablicy zostaje zapisana wartość 0.  
            JCheckBox jc = (JCheckBox) listaPolWyboru.get(j + (16*i));  
            if ( jc.isSelected() ) {  
                listaSciezki[j] = klucz;  
            } else {  
                listaSciezki[j] = 0;  
            }  
        } // koniec pętli wewnętrznej  
  
        utworzSciezke(listaSciezki); ← Tworzymy zdarzenia dla danego instrumentu i wszystkich szesnastu taktów, po czym zapisujemy je na ścieżce.  
        sciezka.add(tworzZdarzenie(176,1,127,0,16));  
    } // koniec pętli zewnętrznej  
  
    sciezka.add(tworzZdarzenie(192,9,1,0,15)); ← Chcemy się upewnić, że zawsze w ostatnim taktie pojawi się jakieś zdarzenie. W przeciwnym razie program może nie odtworzyć wszystkich taktów przed ponownym rozpoczęciem ścieżki.  
    try {  
  
        sekvenser.setSequence(sekwencja);  
        sekvenser.setLoopCount(sekvenser.LOOP_CONTINUOUSLY); ← Ta metoda pozwala określić ilość powtórzeń, w tym przypadku sekwencja będzie powtarzana w nieskończoność.  
        sekvenser.start();  
        sekvenser.setTempoInBPM(120);  
    } catch(Exception e) {e.printStackTrace();}  
} // koniec metody  
  
public class MojStartListener implements ActionListener {  
    public void actionPerformed(ActionEvent a) {  
        utworzSciezkeIOdtworz(); ← A TERAZ ZACZYNAMY GRAĆ!!!  
    }  
} // koniec klasy wewnętrznej
```

```

public class MojStopListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        sekwenser.stop();
    }
} // koniec klasy wewnętrznej

public class MojTempoGListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        float wspTempa = sekwenser.getTempoFactor();
        sekwenser.setTempoFactor((float)wspTempa * 1.03);
    }
} // koniec klasy wewnętrznej

public class MojTempoDListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        float wspTempa = sekwenser.getTempoFactor();
        sekwenser.setTempoFactor((float)wspTempa * .97);
    }
} // koniec klasy wewnętrznej
}

public void utworzSciezke(int[] lista) {
    for (int i = 0; i < 16; i++) {
        int klucz = lista[i];
        if (klucz != 0) {
            sciezka.add(tworzZdarzenie(144,9,klucz, 100, i));
            sciezka.add(tworzZdarzenie(128,9,klucz, 100, i+1));
        }
    }
}

public static MidiEvent tworzZdarzenie(int plc, int kanal, int jeden, int dwa, int takt) {
    MidiEvent zdarzenie = null;
    try {
        ShortMessage a = new ShortMessage();
        a.setMessage(plc, kanal, jeden, dwa);
        zdarzenie = new MidiEvent(a, takt);
    } catch(Exception e) { e.printStackTrace(); }
    return zdarzenie;
} // koniec metody
} // koniec klasy

```

Pozostałe trzy klasy wewnętrzne dla odbiorców zdarzeń generowanych przez przyciski.

Współczynnik tempa modyfikuje tempo sekwensera o podaną wartość. Domyślnie współczynnik ten ma wartość 1.0, zatem my modyfikujemy go przy każdym kliknięciu o wartość +/- 3%.

Ta metoda tworzy zdarzenia dla wybranego instrumentu określające jego wykorzystanie we wszystkich szesnastu taktach. A zatem do metody można przekazać tablicę typu int dla instrumentu Drum bass, a każdy z jej elementów będzie zawierać klucz danego instrumentu bądź wartość 0. Wartość 0 oznacza, że w danym taktie instrument nie ma grać. Jeśli wartość jest różna od zera, tworzymy zdarzenie i dodajemy je do ścieżki.

Tworzymy zdarzenia NOTE ON oraz NOTE OFF i dodajemy je do ścieżki.

To jest metoda pomocnicza przedstawiona już w poprzednim rozdziale. Nic nowego.



Jakie fragmenty kodu generują poniższe interfejsy?

Pięć spośród sześciu poniższych interfejsów użytkownika zostało wygenerowanych przy użyciu jednego z fragmentów kodu przedstawionych na kolejnej stronie. Dopusz j każdy z fragmentów kodu do wygenerowanego przez niego interfejsu.

?

The figure consists of six numbered windows (1 through 6) arranged around a large question mark in the center. Each window represents a different layout of UI elements:

- Window 1:** A simple window with a single button labeled "tesuji" centered at the top of the main content area.
- Window 2:** A window with a single button labeled "watari" centered at the bottom of the main content area.
- Window 3:** A window with a single button labeled "tesuji" positioned on the left side of the main content area.
- Window 4:** A window with a single button labeled "tesuji" positioned on the right side of the main content area.
- Window 5:** A window with a single button labeled "watari" centered at the top of the main content area, set against a dark gray background.
- Window 6:** A window with a single button labeled "watari" centered at the bottom of the main content area, set against a dark gray background.

Fragmenty kodu

D

```
JFrame ramka = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);
 JButton przycisk = new JButton("tesuji");
 JButton przycisk2 = new JButton("watari");
 ramka.getContentPane().add(BorderLayout.NORTH, panel);
 panel.add(przycisk2);
 ramka.getContentPane().add(BorderLayout.CENTER, przycisk);
```

B

```
JFrame ramka = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);
 JButton przycisk = new JButton("tesuji");
 JButton przycisk2 = new JButton("watari");
 panel.add(przycisk2);
 ramka.getContentPane().add(BorderLayout.CENTER, przycisk);
 ramka.getContentPane().add(BorderLayout.EAST, panel);
```

C

```
JFrame ramka = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);
 JButton przycisk = new JButton("tesuji");
 JButton przycisk2 = new JButton("watari");
 panel.add(przycisk2);
 ramka.getContentPane().add(BorderLayout.CENTER, przycisk);
```

A

```
JFrame ramka = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);
 JButton przycisk = new JButton("tesuji");
 JButton przycisk2 = new JButton("watari");
 panel.add(przycisk);
 ramka.getContentPane().add(BorderLayout.NORTH, przycisk2);
 ramka.getContentPane().add(BorderLayout.EAST, panel);
```

E

```
JFrame ramka = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);
 JButton przycisk = new JButton("tesuji");
 JButton przycisk2 = new JButton("watari");
 ramka.getContentPane().add(BorderLayout.SOUTH, panel);
 panel.add(przycisk2);
 ramka.getContentPane().add(BorderLayout.NORTH, przycisk);
```



Ćwiczenie Rozwiążanie

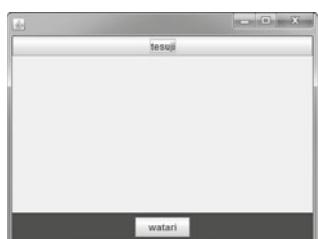
1



2



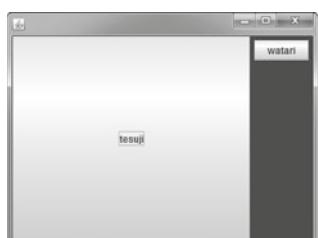
3



4



6



C

```
JFrame ramka = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
 JButton przycisk = new JButton("tesuji");
 JButton przycisk2 = new JButton("watari");
 panel.add(przycisk2);
ramka.getContentPane().add(BorderLayout.CENTER, przycisk);
```

D

```
JFrame ramka = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
 JButton przycisk = new JButton("tesuji");
 JButton przycisk2 = new JButton("watari");
ramka.getContentPane().add(BorderLayout.NORTH, panel);
panel.add(przycisk2);
ramka.getContentPane().add(BorderLayout.CENTER, przycisk);
```

E

```
JFrame ramka = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
 JButton przycisk = new JButton("tesuji");
 JButton przycisk2 = new JButton("watari");
ramka.getContentPane().add(BorderLayout.SOUTH, panel);
panel.add(przycisk2);
ramka.getContentPane().add(BorderLayout.NORTH, przycisk);
```

A

```
JFrame ramka = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
 JButton przycisk = new JButton("tesuji");
 JButton przycisk2 = new JButton("watari");
 panel.add(przycisk);
ramka.getContentPane().add(BorderLayout.NORTH, przycisk2);
ramka.getContentPane().add(BorderLayout.EAST, panel);
```

B

```
JFrame ramka = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
 JButton przycisk = new JButton("tesuji");
 JButton przycisk2 = new JButton("watari");
 panel.add(przycisk2);
ramka.getContentPane().add(BorderLayout.CENTER, przycisk);
ramka.getContentPane().add(BorderLayout.EAST, panel);
```

14. Serializacja i operacje wejścia-wyjścia na plikach

Zapisywanie obiektów



Jeśli będę musiała przeczytać jeszcze jeden plik danych, to myślę, że go zabiję. Doskonale wie, że mogę zapisywać całe obiekty, ale czy mi na to pozwala? NIE – bo to by było zbyt proste. Dobrze, zobaczymy, jak się poczuje, kiedy zrobię...

Obiekty można „pakować”, a następnie odtwarzać.

Obiekty mają swój stan i działanie. Działanie jest określane przez samą klasę obiektu, jednak stan określają poszczególne obiekty.

Co się zatem dzieje, kiedy nadejdzie czas zapisania stanu obiektu? Jeśli piszesz grę, to zapewne będzie Ci potrzebna opcja Zapisz/Odtwórz. Jeśli piszesz aplikację tworzącą wykresy, to zapewne będzie Ci potrzebna opcja Zapisz/Otwórz dane. Jeśli Twój program musi zapisywać stan, to możesz to zrobić w złożony sposób — odczytując stan każdego obiektu i pracowicie zapisując wartość każdej składowej w pliku, nadając jej określony format. Możesz także zrobić to samo w prosty obiektowy sposób — po prostu „zapisać-zamrozić-wysuszyć-zachować” sam obiekt, a następnie go „odczytać-odmrozić-namoczyć-odtworzyć”. Jednak czasami i tak będziesz to musiał zrobić sam w ten bardziej skomplikowany sposób, zwłaszcza jeśli plik zapisywany przez Twój program musi być odczytywany także przez inne aplikacje, które nie zostały napisane w Javie. Dlatego też w tym rozdziale opiszemy oba rozwiązania.

Odczytywanie taktów

Stworzyłeś idealną kompozycję. Teraz chciałbyś ją **zapisać**. Móglbyś sięgnąć po kawałek papieru i zacząć ją ręcznie zapisywać, lecz zamiast tego klikasz przycisk **Zapisz** (lub wybierasz opcję o tej samej nazwie z menu *Plik*). Następnie podajesz nazwę pliku, wybierasz folder i możesz odetchnąć z ulgą, wiedząc, że Twoje dzieło nie zginie bezpowrotnie wraz z niebieskim ekranem śmierci systemu operacyjnego.

Istnieje wiele sposobów zapisania stanu programu napisanego w Javie, a to, który spośród nich wybierzesz, będzie zapewne zależało od sposobu, w jaki planujesz *wykorzystać* zapisane informacje. Poniżej przedstawiliśmy rozwiązania, którymi zajmiemy się w tym rozdziale.

Jeśli dane będą używane wyłącznie przez napisany w Javie program, który je wygenerował:

1 Zastosuj mechanizm serializacji

Zapisz plik zawierający spakowane (serializowane) obiekty. Następnie dodaj do programu możliwość odczytywania serializowanych danych z pliku, odtwarzania ich i zmieniania w żywe, oddychające obiekty zamieszczające sterty.

Jeśli dane będą używane przez inne programy:

2 Zapisz stan obiektów w zwykłym pliku tekstowym

Utwórz plik i oddzielaj zapisywane w nim informacje przy użyciu separatorów, które będą mogły przetworzyć inne programy.

Na przykład stwórz plik z danymi rozdzielanymi znakami tabulacji, który będą mogły odczytywać arkusze kalkulatory i programy obsługujące bazy danych.

Oczywiście to nie są wszystkie dostępne opcje. Dane można zapisywać w dowolnie wybranym formacie. Na przykład zamiast zapisywania danych w czytelnej formie wykorzystującej znaki, można je zapisywać jako bajty. Można także zapisywać dane podstawowych typów Javy w sposób charakterystyczny dla ich typów — istnieją bowiem metody zapisujące liczby całkowite typu int, typu long, wartości logiczne i tak dalej. Niemniej jednak, niezależnie od wykorzystanej metody, podstawowe techniki wykonywania operacji wejścia-wyjścia są do siebie podobne — dane są zapisywane w *czymś*, a to „coś” zazwyczaj jest plikiem przechowywanym na dysku lub strumieniem pochodząącym z połączenia sieciowego. Odczyt danych jest tym samym procesem realizowanym w odwrotnym kierunku — sprowadza się on do odczytania danych z pliku przechowywanego na dysku lub z połączenia sieciowego. Poza tym, wszystko o czym będziemy pisać w tym rozdziale odnosi się do programów, które nie korzystają z baz danych.

Zapisywanie stanu

Wyobraź sobie, że dysponujesz programem, na przykład fantastyczną grą przygodową, której zakończenie wymaga więcej niż jednej sesji. Wraz z rozwojem gry występujące w niej postacie stają się mocniejsze, słabsze, mądrzejsze itd., a dodatkowo zdobywają i używają (jak również gubią) bronie. Nie chcesz zaczynać od samego początku za każdym razem, gdy uruchomisz grę — całe wieki zajęło Ci optymalne przygotowanie swoich postaci do ostatecznej bitwy. Dlatego też potrzebujesz sposobu zapisania stanu swoich postaci oraz odtworzenia go podczas kolejnego uruchamiania gry. A poza tym, ponieważ jesteś także programistą tworzącym tę grę, chciałbyś, aby cała operacja zapisu i odczytu danych była możliwie jak najprostsza (i odporna na wszelkie błędy).

1 Rozwiązywanie pierwsze

Zapisanie trzech serializowanych obiektów postaci w pliku.

Stwórz plik i zapisz w nim trzy serializowane obiekty postaci występujących w grze. Jeśli otworzysz ten plik i spróbujesz odczytać jego zawartość, okaże się, że nie ma ona żadnego sensu, na przykład:

```
..IsrBohater
..%ge8MUImocLjava/lang/
String;[broniet[Ljava/lang/
String;xp2tfur[Ljava.lang.String;"VÁ
E{GxpHuktmiecztpyłsq~>`Troluq~tpięś
ci wielki topór sq~xtMagikuq~tcza
rytniewidzialność
```

2 Rozwiązywanie drugie

Zapisz dane w zwyczajnym pliku tekstowym

Utwórz plik i zapisz w nim trzy wiersze tekstu
- po jednym dla każdej z postaci; informacje umieszczone w wierszu oddziel od siebie przecinkami:

```
50,Elf,łuk,miecz,pył
200,Trol,pięści,wielki topór
120,Magik,czary,niewidzialność
```

Wyobraź sobie, że dysponujesz trzema postaciami, których stan chcesz zapisać...

Bohater
int moc
String typ
Bron[] bronie
wezBron()
uzwijBroni()
zwiększMoc ()
// kolejne metody

moc: 50
typ: Elf
bronie: tuk,
miecz, pył

Obiekt

moc: 200
typ: Trol
bronie: pięści,
wielki topór

Obiekt

moc: 150
typ: Magik
bronie: czary,
niewidzialność

Obiekt

Dla ludzi odczyt serializowanych plików będzie bardzo trudny, jednak dla programów odtwarzanie stanu trzech obiektów z takiego pliku będzie znacznie łatwiejsze (i bezpieczniejsze) niż odczytywanie ich z wartości zapisanych w zwyczajnym pliku tekstowym. Na przykład spróbuj sobie wybrać wszelkie potencjalnie możliwe błędy związane z pomyleniem kolejności odczytywania wartości z takiego pliku! Na przykład sktadowa „typ” mogłaby uzyskać wartość „pył” zamiast „Elf”, a wartość „Elf” stądby się nagle nazwać bronią...

Zapisywanie serializowanego obiektu do pliku

Poniżej przedstawione zostały etap serializacji (czyli zapisywania) obiektu. Nie musisz się zbytnio starać, aby je już teraz zapamiętać — w dalszej części rozdziału zajmiemy się tym zagadnieniem bardziej szczegółowo.

Jeśli plik „MojaGra.ser” nie istnieje,
zostanie automatycznie utworzony.

1 Utwórz strumień FileOutputStream:

```
FileOutputStream strumienPlk = new FileOutputStream("MojaGra.ser");
```

Tworzymy obiekt `FileOutputStream`. Obiekt ten „wie”, jak połączyć się z plikiem (i, w razie konieczności, jak go stworzyć).

2 Utwórz strumień ObjectOutputStream:

```
ObjectOutputStream so = new ObjectOutputStream(strumienPlk);
```

Tworzymy obiekt `ObjectOutputStream`, który pozwala na zapisywanie obiektów, lecz nie potrafi operować na plikach. Potrzebny jest mu zatem „pomocnik”. Rozwiążanie to nosi nazwę tworzenia łańcucha strumieni.

3 Zapisz obiekty:

```
so.writeObject(postac1);  
so.writeObject(postac2);  
so.writeObject(postac3);
```

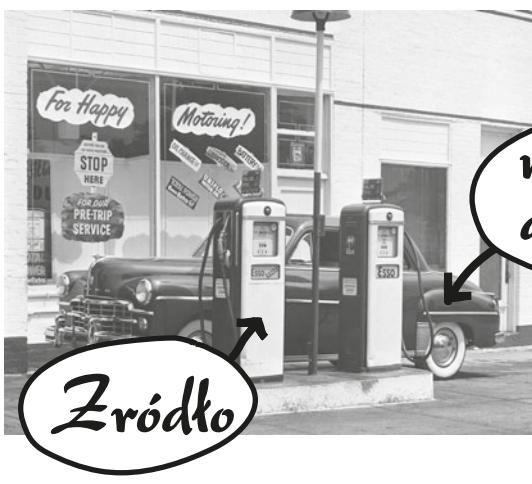
Przeprowadzamy serializację obiektów określonych za pomocą odwołań `postac1`, `postac2` oraz `postac3` i zapisujemy je w pliku „`MojaGra.ser`”.

4 Zamknij strumień ObjectOutputStream:

```
so.close();
```

Zamknięcie strumienia znajdującego się na samym początku łańcucha powoduje zamknięcie wszystkich strumieni znajdujących się za nim; a zatem, strumień `FileOutputStream` (oraz używany przez niego plik) zostanie automatycznie zamknięty.

Dane są przesyłane strumieniami z miejsca na miejsce



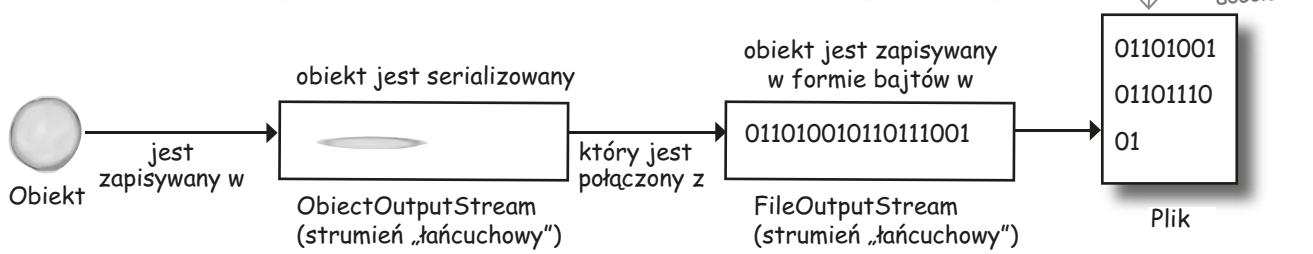
Strumienie połączeniowe reprezentują połączenie pomiędzy źródłem a miejscem docelowym informacji (plikiem, połączeniem sieciowym itp.), natomiast strumienie łańcuchowe nie są w stanie samodzielnie nawiązać połączenia i dlatego muszą korzystać z możliwości strumieni połączeniowych.

Interfejs programistyczny do obsługi operacji wejścia-wyjścia — Java I/O API — zawiera strumienie **połączeniowe**, reprezentujące połączenie z miejscami docelowymi oraz źródłami takimi jak pliki i gniazda sieciowe oraz strumienie **łańcuchowe**, które mogą działać wyłącznie w połączeniu z innymi strumieniami.

Czasami możliwość wykonania jakiejś przydatnej operacji uzyskujemy dopiero po połączeniu co najmniej dwóch strumieni, z których *pierwszy* obsługuje połączenie, a *drugi* udostępnia metody, z których chcemy skorzystać. Dlaczego dwa? Ponieważ w przeważającej większości przypadków strumienie „połączeniowe” działają na zbyt niskim poziomie. Na przykład `FileOutputStream`, będący strumieniem „połączeniowym”, udostępnia metody pozwalające na zapisywanie *bajtów!* Jednak my chcemy zapisywać *obiekty*, potrzebny jest nam zatem strumień **łańcuchowy** działający na wyższym poziomie.

No dobrze, ale dlaczego nie można było stworzyć jednego strumienia, który robi *dokładnie* to, o co nam chodzi? Czyli pozwala na zapisywanie obiektów, lecz w niewidoczny sposób przekształca je w ciąg bajtów? Pomyśl o zasadach dobrego projektowania zorientowanego obiektowo. Każda klasa jest dobra, ale tylko w *jednej* wąskiej dziedzinie. Klasa `FileOutputStream` zapisuje bajty do pliku. Klasa `ObjectOutputStream` zamienia obiekty na dane, które można zapisać w strumieniu. Dlatego też tworzymy strumień `FileOutputStream`, który pozwala nam zapisywać dane w pliku i łączymy go ze strumieniem `ObjectOutputStream` (strumieniem „łańcuchowym”). Kiedy wywołamy metodę `writeObject()` obiektu `ObjectOutputStream`, to wskazany obiekt zostaje umieszczony w strumieniu i przekazany do `FileOutputSteam`, który zapisze bajtową reprezentację obiektu w pliku.

Możliwość łączenia przeróżnych strumieni „połączeniowych” i „łańcuchowych” zapewnia ogromną elastyczność! Gdybyś musiał korzystać tylko z *jednej* klasy obsługującej strumienie, znalazłybyś się na łasce projektantów API i musiałbyś mieć nadzieję, że pomyśleli oni o *wszystkich* operacjach, które kiedykolwiek mógłbyś chcieć wykonać. Jednak dzięki możliwości łączenia strumieni w razie potrzeby możesz stworzyć taką ich kombinację, która zaspokoi *Twoje* potrzeby.



Co tak naprawdę dzieje się z obiektem podczas jego serializacji?

1 Obiekt na stercie

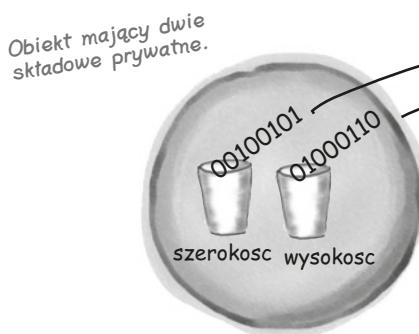


Obiekty na stercie mają stan — czyli wartości składowych. To właśnie dzięki tym wartośćom jeden obiekt danej klasy różni się od innych obiektów tej samej klasy.

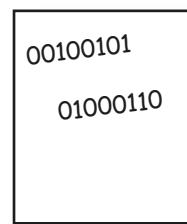
2 Obiekt serializowany



Obiekt serializowany to **zapisane wartości składowych**, na podstawie których na stercie można utworzyć identyczny obiekt.



Wartości są wysysane z obiektu i wtfaczane do strumienia.



Wartości składowych szerokości i wysokości zostały zapisane w pliku „Test.ser” wraz z niewielką ilością pewnych dodatkowych informacji, których wirtualna maszyna Java potrzebuje, aby odtworzyć obiekt.

```
FileOutputStream fs = new FileOutputStream("Test.ser");
ObjectOutputStream os = new ObjectOutputStream(fs);
os.writeObject(mojTest);
```

```
Test mojTest = new Test();
mojTest.setSzerokosc(37);
mojTest.setWysokosc(70);
```

Tworzymy strumień `fileOutputStream`, który nawiązuje połączenie z plikiem „Test.ser”, następnie do tego strumienia dążącamy kolejny — `ObjectOutputStream` — a potem zapisujemy obiekt, wykorzystując w tym celu drugi ze strumieni.

Jednak czym tak naprawdę JEST stan obiektu?

Co trzeba zapisać?

Teraz dochodzimy do naprawdę interesujących rzeczy. Zapisanie wartości jednego z typów podstawowych nie jest żadnym wyzwaniem. Co się jednak dzieje w sytuacji, gdy obiekt posiada składową, która jest odwołaniem do obiektu? Co z obiektem, który ma pięć składowych będących odwołaniami do innych obiektów? Co się dzieje, gdy składowe obiektu mają swoje własne składowe?

Zastanów się nad tym. Jaka część obiektu może być unikalna? Postaraj się wyobrazić sobie, co trzeba odtworzyć, aby można było uzyskać obiekt identyczny z obiektem, który został zapisany. Nowy obiekt zostanie umieszczony w innym miejscu pamięci — to oczywiste — jednak to nas nie obchodzi. Obchodzi nas tylko to, że gdzieś tam, na stercie istnieje obiekt o takim samym stanie co obiekt, który wcześniej zapisaliśmy.



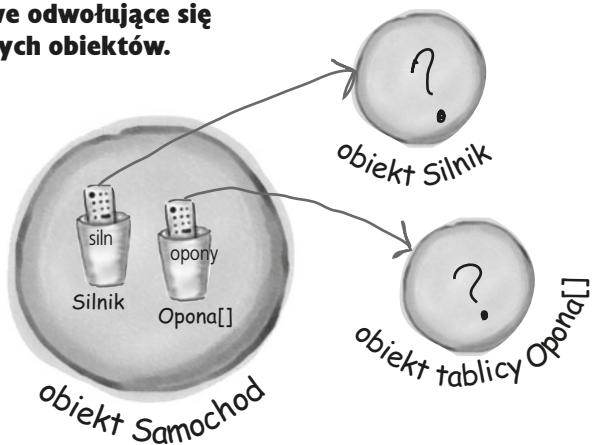
WYŁĘŻ UMYSŁ

Co trzeba zrobić z obiektem Samochod, aby zapisać go w sposób, który pozwoli na późniejsze przywrócenie go do stanu początkowego?

Zastanów się, co będzie Ci potrzebne, aby zapisać obiekt Samochod i jak tę operację wykonać.

I co się stanie, jeśli obiekt Silnik będzie zawierać odwołanie do obiektu Rozrusznik? No i co się dzieje z zawartością tablicy Opona[]?

Obiekt Samochod zawiera dwie składowe odwołujące się do dwóch innych obiektów.



Jakich operacji wymaga zapisanie stanu obiektu Samochod?

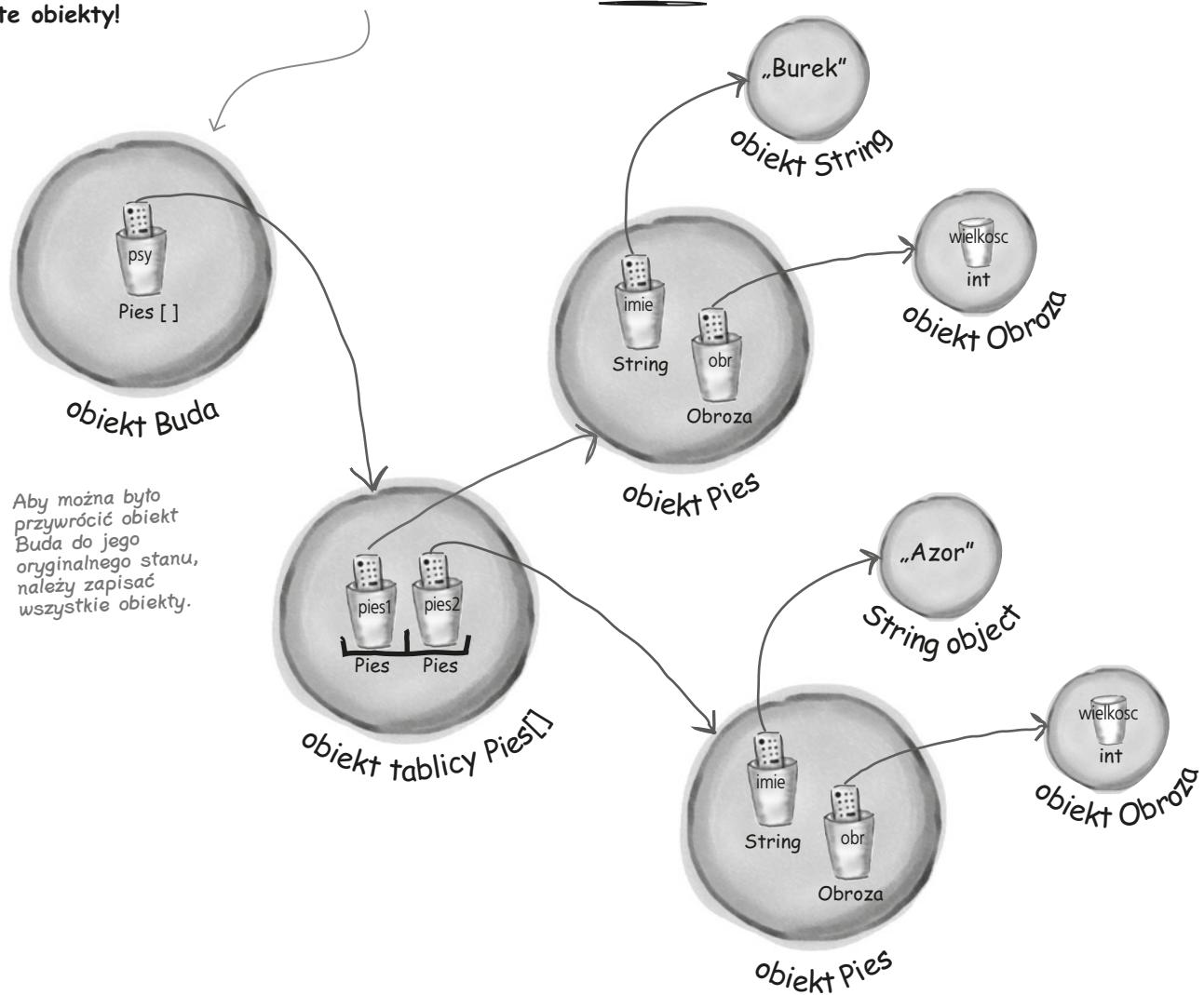
Serializacja obiektów

W przypadku serializacji obiektu, jeśli jego składowe odwołują się do jakichkolwiek innych obiektów, to także te „inne” obiekty zostają serializowane. Jak również wszystkie obiekty, do których odwołują się te „inne” obiekty. A najlepsze jest to, że cały ten proces odbywa się automatycznie!

Przedstawiony poniżej obiekt Buda zawiera odwołanie do tablicy obiektów Pies. Tablica Pies[] zawiera odwołania do dwóch obiektów Pies. Każdy z obiektów Pies zawiera odwołania do obiektu String oraz Obroza. Obiekt String zawiera pewien zbiór znaków, a obiekt Obroza – składową typu int.

W wyniku procesu serializacji zapisany zostaje cały **graf obiektów**. Wszystkie obiekty, do których odwołują się składowe, jak również sam serializowany obiekt.

Podczas zapisywania obiektu Buda zapisane zostają wszystkie te obiekty!



Jeśli chcesz, aby klasa zapewniała możliwość serializacji, zaimplementuj w niej interfejs Serializable

Interfejs Serializable stanowi przykład tak zwanego interfejsu znacznika, gdyż nie zawiera on żadnej metody, którą trzeba by zaimplementować. Jedynym celem istnienia interfejsu Serializable jest to, że dana klasa daje możliwość serializacji obiektów. Innymi słowy, obiekty tego typu można zapisywać przy wykorzystaniu mechanizmu serializacji. Jeśli jakakolwiek klasa bazowa pozwala na serializację, to także jej klasa potomna będzie dysponować tą możliwością, nawet jeśli nie zostanie w niej jawnie zadeklarowany interfejs Serializable. (Właśnie w taki sposób zawsze działają interfejsy. Jeśli klasa bazowa spełnia warunki testu JEST, to także klasa potomna je spełnia).

```
objectOutputStream.writeObject(mojePudelko);
```

Jakikolwiek obiekt zostanie tu umieszczony, jego klasa MUSI implementować interfejs Serializable; w przeciwnym razie podczas działania programu wystąpi błąd.

```
import java.io.*;           ← Interfejs Serializable wchodzi w skład pakietu java.io, zatem musisz go zimportować.

public class Pudelko implements Serializable {           ← Nie trzeba implementować żadnych metod, jednak samo dodanie do deklaracji metody słów „implements Serializable” stanowi sygnał dla JVM, że: „można przeprowadzać serializację obiektów tego typu”.

    private int szerokosc;                                ← Te dwie wartości będą zapisywane.
    private int wysokosc;

    private void setSzerokosc(int s) {
        szerokosc = s;
    }

    private void setWysokosc(int w) {
        wysokosc = w;
    }

    public static void main(String[] args) {

        Pudelko mojePudelko = new Pudelko();
        mojePudelko.setSzerokosc(50);
        mojePudelko.setWysokosc(20);                         ← Nawiązujemy połączenie z plikiem „pudelko.ser”, jeśli taki istnieje. Jeśli plik o podanej nazwie nie istnieje, to zostanie utworzony.

        try {                                                 ← Operacje wejścia-wyjścia mogą zgłaszać wyjątki.
            FileOutputStream fs = new FileOutputStream("pudelko.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(mojePudelko);
            os.close();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Tworzymy strumień ObjectOutputStream połączony ze strumieniem FileOutputStream. Następnie nakazujemy zapisać obiekt w strumieniu.

Serializacja obiektów

Serializacja to operacja typu „wszystko albo nic”.

Czy wyobrażasz sobie, co by się stało, gdyby część stanu obiektu nie została poprawnie zapisana?



Fujjjjj! Skręca mnie na samą myśl o czymś takim! Na przykład gdyby Pies został odtworzony bez żadnej wagi. Albo bez uszu. Albo gdyby po odtworzeniu okazało się, że obroża ma rozmiar 3 zamiast 30. To absolutnie niedopuszczalne!

Albo cały graf obiektu zostanie poprawnie serializowany, albo cała operacja serializacji się nie powiedzie.

Nie można serializować obiektu Staw, jeśli umieszczone w nim obiekty Kaczka nie pozwolą na serializację (ponieważ nie został w nich zaimplementowany interfejs Serializable).

```
import java.io.*;  
  
public class Staw implements Serializable {  
    private Kaczka kaczka = new Kaczka();  
  
    public static void main(String[] args) {  
        Staw mojStaw = new Staw();  
        try {  
            FileOutputStream fs = new FileOutputStream("staw.ser");  
            ObjectOutputStream os = new ObjectOutputStream(fs);  
  
            os.writeObject(mojStaw);  
            os.close();  
        } catch (Exception ex) {  
            ex.printStackTrace();  
        }  
    }  
  
    public class Kaczka {  
        // tu znajduje się kod  
        // klasy Kaczka  
    }  
}
```

Obiekt Staw można serializować
Klasa Staw posiada składową Kaczka.
Podczas serializacji zmiennej mojStaw (obiektu klasy Staw) automatycznie jest serializowana także jej składowa klasy Kaczka.
Ups!! Ale obiektu Kaczka nie można serializować. Klasa Kaczka nie implementuje interfejsu Serializable, zatem próba serializacji obiektu klasy Staw zakończy się niepowodzeniem, gdyż nie będzie można zapisać jego składowej klasy Kaczka.

Oto wyniki, jakie uzyskasz, próbując uruchomić klasę Staw:

```
Wiersz poleceń  
javac Staw  
java Staw  
java.io.NotSerializableException: Kaczka  
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1054)  
    at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:  
1138)  
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:  
952)  
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:  
1052)  
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1022)  
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:278)  
    at Staw.main(Staw.java:13)
```



A zatem czy to jest sytuacja beznadziejna? Czy mam problem, skoro jakiś idiota, który tworzył klasę mojej składowej, zapomniał zaimplementować w niej interfejs Serializable?

Jeśli składowa nie może (lub nie powinna) być zapisywana, oznacz ją słowem kluczowym transient.

Jeśli chcesz, aby jakieś składowe zostały pominięte podczas procesu serializacji obiektu, oznacz je przy użyciu słowa kluczowego **transient**.

To słowo kluczowe oznacza „nie zapisuj tej składowej podczas serializacji, po prostu ją pominij”.

```
import java.net.*;
class Pogawedka implements Serializable {
    transient String idSesji;
    String nazwaUzytkownika;
    // dalsza część kodu
}
```

Składowa nazwaUzytkownika zostanie zapisana podczas procesu serializacji jako część stanu obiektu.

Jeśli masz składową, której nie można, zapisać gdyż nie można jej serializować, możesz ją oznaczyć słowem kluczowym **transient**, a proces serializacji po prostu ją pominie.

Ale z jakich powodów składowa może nie być serializowana? Być może twórca klasy po prostu *zapomniał* zaimplementować w klasie interfejsu **Serializable**. Być może obiekt bazuje na informacjach dostępnych wyłącznie w czasie wykonywania i po prostu nie może być serializowany. Choć większość obiektów w Javie można serializować, to jednak niektóre z nich nie udostępniają takiej możliwości, są to, na przykład: połączenia sieciowe, wątki czy też obiekty plików. Wszystkie te obiekty są zależne od (i unikalne dla) konkretnego „wykonania” i związanej z nim sytuacji. Innymi słowy, obiekty te tworzone są w sposób, który jest unikalny dla konkretnego uruchomienia programu, na konkretnej platformie systemowej i przy użyciu konkretnej wirtualnej maszyny Javy. Po zakończeniu programu nie można w żaden sensowny sposób odtworzyć tych obiektów — za każdym razem trzeba je tworzyć od nowa.

Nie istnieja głupie pytania

P: Jeśli serializacja jest tak ważna, to dlaczego możliwość serializacji nie jest domyślnie dostępna we wszystkich klasach? Dlaczego interfejs `Serializable` nie jest implementowany bezpośrednio w klasie `Object`, dzięki czemu wszystkie jej klasy potomne automatycznie dawałyby możliwość serializacji?

O: Nawet jeśli większość klas będzie implementować interfejs `Serializable` i powinna to robić, to jednak wciąż masz możliwość wyboru. I musisz świadomie podejmować decyzję o „włączeniu” możliwości serializacji dla każdej z tworzonych klas — właśnie poprzez jawną implementację interfejsu `Serializable`. Przede wszystkim, gdyby możliwość serializacji była domyślnie „włączona”, to w jaki sposób można by ją „wyłączyć”. W końcu interfejsy oznaczają dostępność pewnych możliwości funkcjonalnych, a nie ich brak. Dlatego też mechanizm polimorfizmu nie działałby poprawnie, gdybyś musiał informować o braku możliwości zapisu klasy poprzez stwierdzanie „implementuj `NonSerializable`”.

P: Dlaczego miałbym kiedykolwiek tworzyć klasę, której nie będzie można serializować?

O: Istnieje bardzo niewiele takich powodów. Ale, na przykład z powodów bezpieczeństwa, możesz nie życzyć sobie zapisywania obiektu hasła. Możesz także używać obiektów, których zapisywanie po prostu nie ma sensu, gdyż nie można zapisywać ich kluczowych składowych. W takim przypadku także nie będzie żadnego sensownego sposobu serializacji Twojej klasy.

P: Jeśli klasa, której używam, nie pozwala na serializację, choć jest to nieuzasadnione (chyba że jej twórca po prostu zapomniał o zaimplementowaniu interfejsu `Serializable`), to czy mogę stworzyć jej klasę potomną i w niej zaimplementować niezbędny interfejs?

O: Tak! Jeśli tylko można stworzyć klasę potomną (czyli jeśli klasa, którą chcesz rozszerzać, nie jest finalna), to nic nie stoi na przeszkodzie, abyś zapewnił możliwość serializacji w swojej klasie potomnej, a następnie używał jej obiektów wszędzie tam, gdzie są oczekiwane obiekty klasy bazowej. (Pamiętaj, że polimorfizm na to pozwala). Jednak przykład ten nasuwa ciekawe pytanie: Co to oznacza, że nie można serializować kasy?

P: Skoro o tym wspominacie... Jak należy interpretować sytuację, w której klasy bazowej nie można serializować, a jej klasę potomną można?

O: Przede wszystkim musimy zobaczyć, co się dzieje podczas deserializacji (a tym zagadnieniem zajmiemy się na kilku kolejnych stronach). Ogólnie rzeczą biorąc, kiedy obiekt jest deserializowany, a jego klasa bazowa *nie* pozwala na serializację, to zostanie wywołany konstruktor klasy bazowej, zupełnie jak gdyby był tworzony nowy obiekt tej klasy. Jeśli nie ma żadnego ważnego powodu, aby klasa nie zapewniała możliwości serializacji, to stworzenie jej klasy potomnej, która zapewnia taką możliwość, jest dobrym rozwiązaniem.

P: Ojej! Właśnie zdałem sobie sprawę z czegoś ważnego... Jeśli oznaczmy składową przy użyciu słowa kluczowego `transient`, to jej wartość zostanie pominięta podczas serializacji. A co się z nią potem dzieje? Rozwiązałismy już problem występujący podczas serializacji takich składowych, ale czy nie będziemy ich potrzebować po odtworzeniu obiektu? Innymi słowy, czy cały sens serializacji nie polegał właściwie na zachowaniu stanu obiektu?

O: Tak, to jest problem. Ale na szczęście można go rozwiązać. W przypadku odtwarzania serializowanego obiektu wszystkie jego składowe zawierające odwołania

i oznaczone słowem kluczowym `transient` uzyskają wartość `null`, i to niezależnie od tego, jaka była ich wartość podczas zapisywania obiektu. Oznacza to, że cały graf obiektów skojarzony z konkretną składową nie zostanie zapisany. To oczywiście nie jest dobra sytuacja, gdyż prawdopodobnie składowa ta powinna mieć wartość inną niż `null`.

Istnieją dwa rozwiązania tego problemu:

1. Podczas odtwarzania obiektu należy ponownie zainicjować te składowe i przypisać im jakieś wartości domyślne. Rozwiązań to można zastosować, jeśli stan odtwarzanego obiektu nie zależy od wartości składowej oznaczonej słowem kluczowym `transient`. Innymi słowy, zapewne ważne będzie, aby obiekt `Pies` zawierał obiekt `Obroza`, jednak prawdopodobnie wszystkie obiekty `Obroza` są takie same, zatem nie ma znaczenia, czy odtworzony `Pies` będzie mieć stary czy też zupełnie nowy obiekt `Obroza`. I tak nikt nie zauważa różnicy.

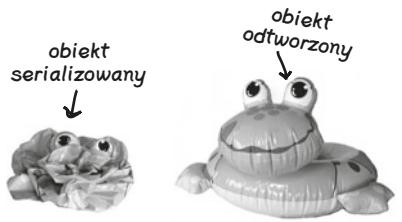
2. Jeżeli wartość składowej oznaczonej słowem kluczowym `transient` ma znaczenie (na przykład, jeśli wzór i kolor obiektu `Obroza` jest unikalny dla każdego obiektu `Pies`), to musisz zapisać kluczowe aspekty takiej składowej, a później użyć ich do stworzenia nowej wartości, która będzie identyczna z oryginalną. (Na przykład musisz zapisać wartości składowych określających wzór i kolor obrozy, a następnie użyć ich do stworzenia nowego obiektu `Obroza`).

P: Co się dzieje, gdy dwa obiekty w grafie zapisywanyego obiektu są takie same? Na przykład, jeśli w obiekcie `Buda` zapisane są dwa obiekty `Kot`, a oba te obiekty zawierają odwołania do tego samego obiektu `Własciciel`? Czy w takiej sytuacji obiekt `Własciciel` zostanie zapisany dwa razy? Mam nadzieję, że nie.

O: Doskonałe pytanie! Na szczęście mechanizm serializacji jest na tyle mądry, iż wie, kiedy dwa obiekty w grafie są takie same. W takim przypadku tylko jeden z nich zostaje zapisany, a podczas deserializacji odtwarzane są wszystkie odwołania do tego jednego obiektu.

Deserializacja — odtwarzanie obiektów

Cała idea serializacji polega na możliwości odtworzenia obiektu w jego oryginalnej postaci po pewnym czasie i podczas innego „uruchomienia” wirtualnej maszyny Javy (która w ogóle może być inną maszyną niż ta, która była używana w czasie serializacji obiektu). Deserializacja jest bardzo podobna do serializacji z tym, że przebiega w odwrotnym kierunku.



1 Utwórz strumień `FileInputStream`

```
FileInputStream strumienPlk = new FileInputStream("MojaGra.ser");
```

Jeśli plik „MojaGra.ser” nie istnieje,
zostanie zgłoszony wyjątek.

Tworzymy obiekt `FileInputStream`.
Obiekt tej klasy wie, jak należy
otworzyć połączenie z plikiem.

2 Utwórz strumień `ObjectInputStream`

```
ObjectInputStream os = new ObjectInputStream(strumienPlk);
```

Strumień `ObjectInputStream` potrafi odczytywać
obiekty, lecz nie wie, jak nawiązać połączenie
z plikiem. Należy go zatem „wspomóc”,
przekazując do niego jakiś strumień
„połączeniowy” — w tym przypadku
jest to strumień `FileInputStream`.

3 Odczytaj obiekty

```
Object obj1 = os.readObject();
Object obj2 = os.readObject();
Object obj3 = os.readObject();
```

Każde wywołanie metody `readObject()` powoduje
odczytanie kolejnego obiektu ze strumienia.
A zatem obiekty zostaną odtworzone w takiej samej
 kolejności, w jakiej je zapisałesz. Pamiętaj, że jeśli
spróbujesz odczytać więcej obiektów, niż zapisałesz,
zostanie zgłoszony wyjątek.

4 Wykonaj rzutowanie obiektów

```
Bohater elf = (Bohater) obj1;
Bohater trol = (Bohater) obj2;
Bohater magik = (Bohater) obj3;
```

Wartość zwracana przez metodę
`readObject()` jest typu `Object` (podobnie
jak było w przypadku `ArrayList`),
a zatem musimy rzutować obiekty
na taki typ, jakiego faktycznie są.

5 Zamknij strumień `ObjectInputStream`

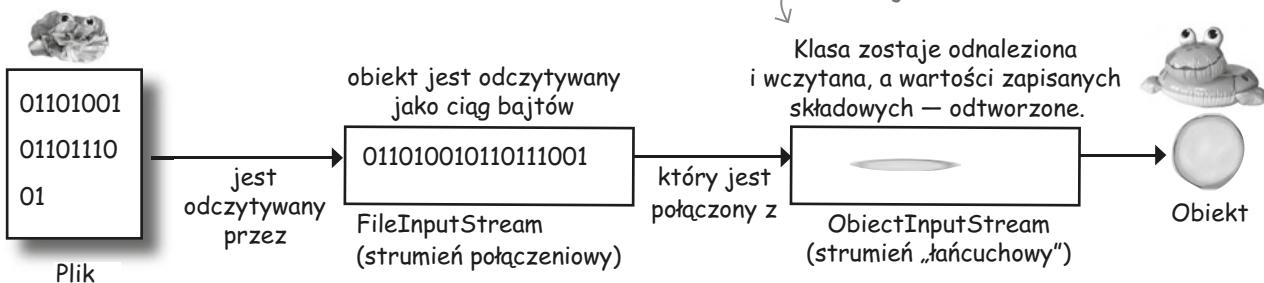
```
os.close();
```

Zamknięcie strumienia znajdującego się na
samym początku łańcucha powoduje zamknięcie
wszystkich strumieni znajdujących się za
nim; a zatem strumień `FileInputStream`
(oraz używany przez niego plik) zostanie
automatycznie zamknięty.

Co się dzieje podczas deserializacji?

Podczas deserializacji obiektu wirtualna maszyna Javy stara się przywrócić go do „życia”; w tym celu tworzy na stercie nowy obiekt mający ten sam stan, który miał oryginalny obiekt w momencie przeprowadzania serializacji. Oczywiście nie dotyczy to składowych oznaczonych jako „transient”, w których zapisywane są wartości null (jeśli składowe są odwołaniami) bądź — w przypadku składowych typów podstawowych — odpowiednie wartości domyślne.

Jeśli wirtualna maszyna Javy nie będzie w stanie znaleźć lub wczytać klasy, to w tym momencie zostanie zgłoszony wyjątek.



- 1 Obiekt jest odczytywany ze strumienia.
- 2 Wirtualna maszyna Javy określa (na podstawie informacji zapisanych w serializowanym obiekcie) typ klasy obiektu.
- 3 Wirtualna maszyna Javy stara się znaleźć i wczytać klasę obiektu. Jeśli klasy nie uda się znaleźć lub wczytać, zgłoszany jest wyjątek, a proces deserializacji kończy się niepowodzeniem.
- 4 Na stercie jest przydzielany obszar dla nowego obiektu, jednak konstruktor serializowanego obiektu NIE jest wywoływany! Oczywiście, gdyby konstruktor został wywołany, to określony domyślny stan „nowego” obiektu, a nie o to nam chodzi. Chcemy, aby obiekt uzyskał stan, jaki miał w chwili przeprowadzania serializacji, a nie w chwili, gdy został utworzony.

- 5 Jeśli gdzieś w drzewie dziedziczenia obiektu znajduje się klasa, która nie pozwala na serializację, to zostanie wywołany jej konstruktor oraz konstruktory wszystkich jej klas bazowych (nawet, jeśli te klasy bazowe pozwalają na serializację). Kiedy rozpocznie się sekwencja wywoływania konstruktorów, nie można jej przerwać, a to oznacza, że ponownie zostanie zainicjowany stan pierwszej klasy, która nie pozwala na serializację, oraz wszystkich jej klas bazowych.
- 6 W składowych są zapisywane wartości odczytane z zapisanego stanu obiektu. W składowych oznaczonych jako „transient” zapisywane są wartości null (jeśli składowe te są odwołaniami) bądź — w przypadku składowych typów podstawowych — odpowiednie wartości domyślne.

Nie istnieją głupie pytania

P: Dlaczego klasa nie zostaje zapisana jako jeden z elementów obiektu? Dzięki temu można by uniknąć problemów z odnajdywaniem klasy.

O: Pewnie, że twórcy Javy mogliby stworzyć mechanizm serializacji działający w taki sposób. Jednak wyobraź sobie to ogromne marnotrawstwo miejsca i czasu. Co prawda rozwiązanie takie nie byłoby zapewne tak niekorzystne w przypadkach zapisywania serializowanych danych na lokalnym dysku twardym, jednak serializowane dane można także przekazywać przy użyciu połączeń sieciowych. Gdyby do każdego serializowanego obiektu była dodawana jego klasa, to przepustowość stałaaby się jeszcze większym problemem, niż aktualnie.

Jeśli chodzi o serializowane obiekty przesypane przez sieć, to tak naprawdę istnieje mechanizm, który dodaje do nich adres URL określający, gdzie można znaleźć ich klasę. Rozwiążanie to jest używane w technologii RMI (ang. Remote Method Invocation — wywoływanie zdalnych metod), dzięki czemu można wysłać serializowany obiekt jako, na przykład, argument wywołania

metody, a wirtualna maszyna Javy odbierająca wywołanie może w razie konieczności pobrać klasę ze wskazanego miejsca i wczytać ją; a co najlepsze — wszystko to dzieje się automatycznie. (Technologię RMI opiszymy w rozdziale 18.).

P: A co ze składowymi statycznymi? Czy one są serializowane?

O: Nie. Pamiętaj, że w tym przypadku dostępna jest „jedna składowa statyczna w klasie”, a nie „po jednej składowej w każdym z obiektów”. Składowe statyczne nie są serializowane, a kiedy obiekt jest odtwarzany, przyjmują one wartość, jaką jest w danej chwili dostępna w klasie. A oto morał: nie uzależniaj serializowanych obiektów od zmiennych statycznych, których wartości mogą się dynamicznie zmieniać! Wartość takiej składowej może być inna w chwili, gdy obiekt będzie odtwarzany.

Zapisywanie i odtwarzanie postaci biorących udział w grze

```

import java.io.*;

public class ArchiwizatorStanuGry {
    public static void main(String[] args) {
        Bohater postac1 = new Bohater(50, "Elf", new String[] {"łuk", "miecz", "pył"});
        Bohater postac2 = new Bohater(200, "Trol", new String[] {"pięści", "wielki topór"});
        Bohater postac3 = new Bohater(120, "Magik", new String[] {"czary", "niewidzialność"});

        // tu wykonujemy jakieś operacje, które mogą zmieniać stan postaci,
        // a następnie zapisujemy te obiekty

        try {
            ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream("Gra.ser"));
            os.writeObject(postac1);
            os.writeObject(postac2);
            os.writeObject(postac3);
            os.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        postac1 = null;          Przypisujemy tym zmiennym
        postac2 = null;          wartości null, żeby
        postac3 = null;          uniemożliwić odwołanie się
                                od obiektów na stercie.
    }

    try {
        ObjectInputStream is = new ObjectInputStream(new FileInputStream("Gra.ser"));
        Bohater p1N = (Bohater) is.readObject();
        Bohater p2N = (Bohater) is.readObject();
        Bohater p3N = (Bohater) is.readObject();

        System.out.println("Typ postaci 1.:" + p1N.getTyp());
        System.out.println("Typ postaci 2.:" + p2N.getTyp()); ← Sprawdzamy, by przekonać się,
        System.out.println("Typ postaci 3.:" + p3N.getTyp());      czy wszystko poszło jak należy.
    } catch (Exception ex) {
        ex.printStackTrace ();
    }
}

```

Wiersz poleceń

```

T:\>java ArchiwizatorStanuGry
Typ postaci 1.:Elf
Typ postaci 2.:Trol
Typ postaci 3.:Magik
T:\>_

```

Utworzymy kilku bohaterów gry...

A teraz odczytujemy bohaterów z pliku.



Klasa bohatera gry

```
import java.io.*;  
  
public class Bohater implements Serializable {  
    int moc;  
    String typ;  
    String[] bronie;  
  
    public Bohater(int m, String t, String[] b) {  
        moc = m;  
        typ = t;  
        bronie = b;  
    }  
  
    public int getMoc() {  
        return moc;  
    }  
  
    public String getTyp() {  
        return typ;  
    }  
  
    public String getBronie() {  
        String listaBroni = "";  
        for (int i = 0; i < bronie.length; i++) {  
            listaBroni += bronie[i] + " ";  
        }  
        return listaBroni;  
    }  
}
```

To prosta klasa, utworzona jedynie na potrzeby przetestowania serializacji, poza tym nie ma żadnej gry... ale tym możesz się zająć samemu, w ramach eksperymentów.

Serializacja obiektów



CELNE SPOSTRZEŻENIA

- Możesz zapisać stan obiektu, przeprowadzając jego serializację.
- W celu wykonania serializacji obiektu będziesz potrzebował strumienia ObjectOutputStream (należącego do pakietu `java.io`).
- Można wyróżnić dwa rodzaje strumieni — „połączniowe” oraz „łańcuchowe”.
- Strumienie „połączniowe” reprezentują połączenie z miejscem docelowym lub ze źródłem danych; zazwyczaj jest to plik, połączenie sieciowe lub konsola.
- Strumienie „łańcuchowe” nie są w stanie nawiązać połączenia z miejscem docelowym lub źródłem danych i dlatego muszą w tym celu korzystać z pomocy strumieni „połączniowych” (lub innych).
- Aby przeprowadzić serializację obiektu, utwórz strumień FileOutputStream, a następnie połącz go ze strumieniem ObjectOutputStream.
- Aby przeprowadzić serializację obiektu, wywołaj metodę `writeObject(obiect)` strumienia ObjectOutputStream. W ogóle nie będziesz musiał posługiwać się metodami strumienia FileOutputStream.
- Aby można było serializować obiekt, jego klasa musi implementować interfejs Serializable. Jeśli jedna z klas bazowych naszej klasy implementuje ten interfejs, to obiekty naszej kasy będzie można serializować nawet, jeśli w naszej klasie interfejs Serializable nie jest jawnie *implementowany*.
- Jeśli obiekt może być serializowany, zapisywany będzie cały graf tego obiektu. To oznacza, że zapisane zostaną wszystkie obiekty, do których odwołują się składowe serializowanego obiektu, obiekty, do których odwołują się składowe tych obiektów, i tak dalej.
- Jeśli nie można przeprowadzić serializacji któregoś z obiektów należących do grafu, to podczas działania programu zostanie zgłoszony wyjątek, chyba że składowa zawierająca odwołanie do tego obiektu zostanie pominięta podczas serializacji.
- Jeśli chcesz, aby składowa została pominięta podczas procesu serializacji, oznacz ją słowem kluczowym `transient`. Po odtworzeniu w składowej zostanie zapisana wartość null (jeśli składowa zawiera odwołanie) lub odpowiednia wartość domyślna (w przypadku składowych typów podstawowych).
- Podczas deserializacji wirtualna maszyna Javy musi mieć dostęp do klas wszystkich obiektów należących do grafu odtwarzanego obiektu.
- Obiekty są odczytywane w takiej samej kolejności, w jakiej zostały zapisane. Do odczytywania obiektów służy metoda `readObject()`.
- Metoda `readObject()` zwraca wartości typu Object, dlatego też obiekty odtworzone w procesie deserializacji należy rzutować na ich faktyczne typy.
- Składowe statyczne nie są serializowane! Nie ma sensu zapisywać wartości składowych statycznych jako elementów stanu konkretnego obiektu, gdyż wszystkie obiekty danego typu korzystają z tej samej wartości składowej statycznej — wartości przechowywanej w danej klasie.

Zapisywanie łańcucha znaków w pliku tekstowym

Zapisywanie obiektów przy wykorzystaniu mechanizmu serializacji jest najprostszym sposobem zapisywania i odtwarzania danych podczas kolejnych uruchomień programu napisanego w Javie, jednak czasami konieczne będzie zapisanie danych w zwyczajnym pliku tekstowym. Wyobraź sobie, że Twój program musi zapisywać dane w prostym pliku tekstowym, który musi być odczytywany przez jakiś inny program (który prawdopodobnie nie został napisany w Javie). Na przykład, możesz używać serwletu (kodu napisanego w Javie i wykonywanego przez serwer WWW), który pobiera informacje wpisane przez użytkownika w przeglądarce i zapisuje je w pliku tekstowym. Taki plik można następnie otworzyć w arkuszu kalkulacyjnym i przeanalizować.

Zapisywanie danych tekstowych (czyli w rzeczywistości obiektów `String`) jest podobne do zapisywania obiektów. Różnice pomiędzy obydwiema operacjami polegają na tym, iż w tym przypadku zapisujemy ciągi znaków, a nie obiekt, a zamiast strumienia `FileOutputStream` jest używany strumień `FileWriter` (oprócz tego nie trzeba dodatkowo używać strumienia `ObjectOutputStream`).

Jak mogłyby wyglądać informacje o postaciach biorących udział w grze, gdyby zostały zapisane przez człowieka — w pliku tekstowym, którego zawartość można odczytać i zrozumieć.

```
50, Elf, łuk, miecz, pył  
200, Troll, pięści, wielki topór  
120, Magik, czary, niewidzialność
```

Aby zapisać serializowany obiekt:

```
objectOutputStream.writeObject(obiekt);
```

Aby zapisać łańcuch znaków:

```
fileWriter.write("Mój pierwszy zapisywany łańcuch znaków");
```

```
import java.io.*; ← Aby móc korzystać z klasy FileWriter,  
                      potrzebny nam będzie pakiet java.io.
```

```
public class ZapisPliku {  
    public static void main(String[] args) {  
        try {  
            FileWriter pisarz = new FileWriter("test.txt");  
            pisarz.write("Witamy w pliku tekstowym!"); ← Metoda write() wymaga przekazania  
                                              łańcucha znaków.  
            pisarz.close(); ← Kiedy już wykonasz wszystkie  
                               operacje — zamknij strumień!  
        } catch (IOException ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

WSZYSTKIE operacje wejścia-wyjścia muszą być umieszczone wewnątrz bloku `try-catch`. Każda z nich może zgłosić wyjątek `IOException`!

Jeśli plik „test.txt” nie będzie istnieć, to strumień `FileWriter` go utworzy.

Zapisywanie pliku tekstowego

Przykład obsługi plików tekstowych — eKarteczki

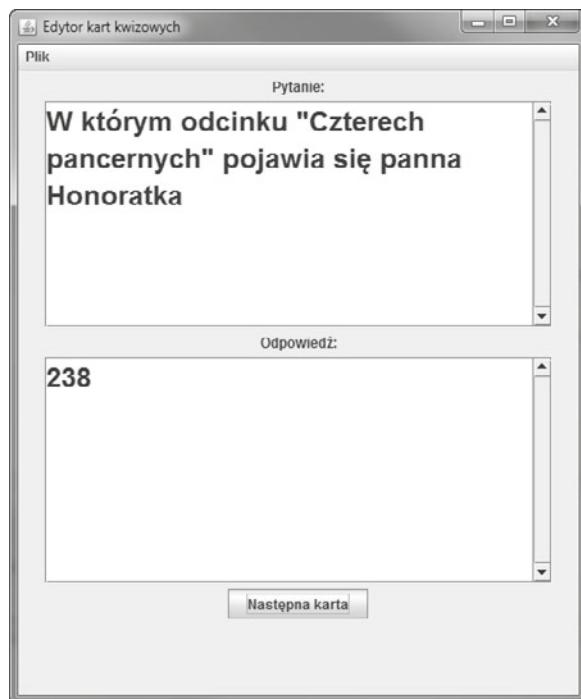
Czy pamiętasz te kwizowe karteczki używane w szkole, gdzie na jednej stronie było zapisane pytanie, a na drugiej odpowiedź? Nie są one szczególnie pomocne, kiedy trzeba coś zrozumieć, jednak nic ich nie zastąpi w przypadkach, gdy trzeba się czegoś nauczyć na pamięć. Kiedy trzeba „wkuc” jakieś fakty. Można je także wykorzystać do zabawy.

Mamy zamiar stworzyć elektroniczną wersję takich karteczek; nasz program będzie się składał z trzech klas:

1. **KartaKwizowaEdytor** — to proste narzędzie służące do tworzenia i zapisywania zbiorów karteczek kwizowych.
2. **KwizGra** — to mechanizm odtwarzający karteczki, który potrafi odczytać zbiór karteczek kwizowych zapisany w pliku i zadawać „zapisane” na nich pytania.
3. **KartaKwizowa** — to prosta klasa reprezentująca dane „zapisane” na jednej karteczce. W dalszej części rozdziału przedstawimy kod pierwszych dwóch klas, natomiast stworzenie klasy KartaKwizowa będzie Twoim zadaniem. Tworząc tę klasę, powinieneś posłużyć się tym schematem

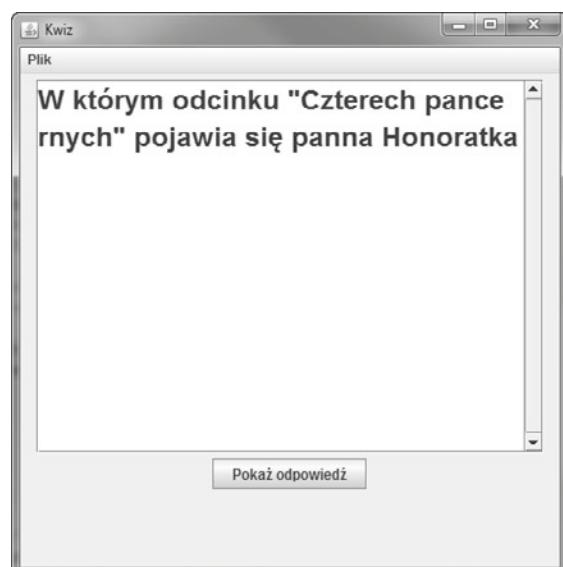


KartaKwizowa
KartaKwizowa(p, o)
pytanie odpowiedź
getPytanie() getOdpowiedz()



KartaKwizowaEdytor

Ten program ma menu *Plik*, a w nim opcję *Zapisz*, która pozwala zapisać bieżący zbiór karteczek w pliku tekstowym.



KwizGra

Posiada menu *Plik* z opcją *Otwórz* pozwalającą na wczytanie zbioru karteczek z pliku źródłowego.

Edytor karteczek kwizowych (zarys kodu)

```

public class KartaKwizowaEdytor {
    public void doDziela() {
        // stworzenie i wyświetlenie graficznego interfejsu użytkownika
    }
    Klasa wewnętrzna
}

public class NastepnaKartaListener implements ActionListener {
    public void actionPerformed(ActionEvent zd) {
        // dodanie bieżącej karteczki do listy i wyczyszczenie wielowierszowych pól tekstowych
    }
}
Klasa wewnętrzna

public class ZapiszMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent zd) {
        // wyświetlenie okna dialogowego obsługi plików
        // użytkownik może podać nazwę pliku i zapisać w nim zbiór karteczek
    }
}
Klasa wewnętrzna

public class NowyMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent zd) {
        // wyczyszczenie listy karteczek oraz wielowierszowych pól tekstowych
    }
}

public void zapiszPlik (File plik) {
    // odczytanie każdego elementu listy karteczek i zapisanie w pliku tekstowym
    // w sposób pozwalający na jego późniejsze odczytanie (czyli z jednoznacznymi separatorami
    // pomiędzy poszczególnymi elementami pliku).
}

```

Metoda tworzy i wyświetla graficzny interfejs użytkownika oraz tworzy i rejestruje odbiorców zdarzeń.

Metoda jest wywoływana w chwili, gdy użytkownik kliknie przycisk Następna karta, co oznacza, że użytkownik chce zapisać na liście bieżącą karteczkę kwizową i utworzyć następną.

Metoda jest wywoływana, gdy użytkownik wybierze z menu Plik opcję Zapisz, co oznacza, że chce zapisać wszystkie karteczki aktualnie dostępne na liście, tworząc z nich jeden „zbiór” (na przykład zbiór mechanika kwantowa, zbiór gwiazdy Hollywood, zbiór Java góra i tak dalej)

Metoda jest wywoywana, gdy użytkownik wybierze opcję Nowy z menu Plik, co oznacza, że użytkownik chce utworzyć zupełnie nowy zbiór karteczek (zatem czyścimy listę karteczek oraz pola tekstowe).

Metoda wywoływana przez klasę ZapiszMenuListener, odpowiedzialną za zapisanie danych w pliku.

Kod klasy KartaKwizowaEdytor

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class KartaKwizowaEdytor {

    private JTextArea pytanie;
    private JTextArea odpowiedz;
    private ArrayList<KartaKwizowa> listaKart;
    private JFrame ramka;

    public static void main(String[] args) {
        KartaKwizowaEdytor edytor = new KartaKwizowaEdytor();
        edytor.doDziela();
    }

    public void doDziela() {
        // utworzenie graficznego interfejsu użytkownika
        ramka = new JFrame("Edytor kart kwizowych");
        ramka.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panelGlowny = new JPanel();
        Font czcionkaDuza = new Font("sanserif", Font.BOLD, 24);
        pytanie = new JTextArea(6, 20);
        pytanie.setLineWrap(true);
        pytanie.setWrapStyleWord(true);
        pytanie.setFont(czcionkaDuza);

        JScrollPane przewijaniePyt = new JScrollPane(pytanie);
        przewijaniePyt.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        przewijaniePyt.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        odpowiedz = new JTextArea(6, 20);
        odpowiedz.setLineWrap(true);
        odpowiedz.setWrapStyleWord(true);
        odpowiedz.setFont(czcionkaDuza);

        JScrollPane przewijanieOdp = new JScrollPane(odpowiedz);
        przewijanieOdp.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        przewijanieOdp.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        JButton przyciskNastepna = new JButton("Następna karta");

        listaKart = new ArrayList<KartaKwizowa>();

        JLabel etykietaPyt = new JLabel("Pytanie:");
        JLabel etykietaOdp = new JLabel("Odpowiedź:");

        panelGlowny.add(etykietaPyt);
        panelGlowny.add(przewijaniePyt);
        panelGlowny.add(etykietaOdp);
        panelGlowny.add(przewijanieOdp);
        panelGlowny.add(przyciskNastepna);
        przyciskNastepna.addActionListener(new NastepnaKartaListener());
        JMenuBar menu = new JMenuBar();
        JMenu menuPlik = new JMenu("Plik");
        JMenuItem opcjaNowa = new JMenuItem("Nowy");
    }
}
```

Na tej stronie przedstawiony został cały kod tworzący graficzny interfejs użytkownika naszego programu. Nie ma w nim nic szczególnego, choć, być może, zechcesz zwrócić uwagę na kod operujący na obiektach `MenuBar`, `Menu` oraz `MenuItem`.

```

JMenuItem opcjaZapisz = new JMenuItem("Zapisz");
opcjaNowa.addActionListener(new NowyMenuListener());
opcjaZapisz.addActionListener(new ZapiszMenuListener());
menuPlik.add(opcjaNowa);
menuPlik.add(opcjaZapisz);
menu.add(menuPlik);
ramka.setJMenuBar(menu);
ramka.getContentPane().add(BorderLayout.CENTER, panelGlowny);
ramka.setSize(500, 600);
ramka.setVisible(true);
}

public class NastepnaKartaListener implements ActionListener {
    public void actionPerformed(ActionEvent zd) {
        KartaKwizowa karta = new KartaKwizowa(pytnie.getText(), odpowiedz.getText());
        listaKart.add(karta);
        czysckarte();
    }
}

public class ZapiszMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent zd) {
        KartaKwizowa karta = new KartaKwizowa(pytnie.getText(), odpowiedz.getText());
        listaKart.add(karta);

        JFileChooser plikDanych = new JFileChooser();
        plikDanych.showSaveDialog(ramka);
        zapiszPlik(plikDanych.getSelectedFile());
    }
}

public class NowyMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent zd) {
        listaKart.clear();
        czysckarte();
    }
}

private void czysckarte() {
    pytnie.setText("");
    odpowiedz.setText("");
    pytnie.requestFocus();
}

private void zapiszPlik(File plik) {
    try {
        BufferedWriter pisarz = new BufferedWriter(new FileWriter(plik));

        for (KartaKwizowa karta : listaKart) {
            pisarz.write(karta.getPytnie() + "/");
            pisarz.write(karta.getOdpowiedz() + "\n");
        }
        pisarz.close();
    } catch (IOException ex) {
        System.out.println("Nie można zapisać pliku kart!");
        ex.printStackTrace();
    }
}
}

```

Tworzymy nowy pasek menu, menu Plik, a następnie dodajemy do tego menu opcje Nowy i Zapisz. Potem dodajemy menu do paska menu i informujemy ramkę, że ma używać tego paska menu. Opcje menu mogą generować zdarzenia ActionEvent.

Ten fragment kodu wyświetla okno dialogowe do obsługi plików (o nazwie Save). Realizacja programu zatrzymuje się na tym wierszu i jest wznowiana dopiero, gdy użytkownik kliknie przycisk Save w oknie dialogowym. Cała obsługa okna dialogowego, wybieranie pliku itp. są realizowane w całości w klasie JFileChooser! To naprawdę jest aż tak proste.

Ta metoda odpowiada za faktyczne zapisanie danych w pliku (i jest wywoływana przez metodę obsługi zdarzeń ZapiszMenuListener). Jej argumentem jest obiekt File, reprezentujący plik, który użytkownik zapisuje. Klasę File przedstawimy dokładniej na następnej stronie.

Łączymy strumień BufferedWriter z FileWriter, aby poprawić efektywność operacji zapisywania danych w pliku. (Tym zagadnieniem zajmiemy się w dalszej części rozdziału).

Pobieramy całą zawartość listy kart i zapisujemy je po kolejno w pliku. Każda karta jest zapisywana w nowym wierszu, pytanie jest oddzielane od odpowiedzi znakiem „/”, a na końcu każdego wiersza umieszczamy znak nowego wiersza („\n”).

Klasa `java.io.File`

Klasa `java.io.File` reprezentuje plik zapisany na dysku, jednak nie reprezentuje jego zawartości. Proszę? Wyobraź sobie obiekt `File` jako coś, co bardziej przypomina ścieżkę dostępu do pliku, lub nawet folderu, niż sam plik wraz z jego zawartością. Na przykład klasa `File` nie posiada metod do zapisu i odczytu danych. Klasa ta jest jednak BARDZO przydatna pod jednym względem — stanowi znacznie bezpieczniejszy sposób reprezentacji pliku niż przechowanie jego nazwy w postaci zwyczajnego łańcucha znaków. Na przykład większość klas, które w swoim konstruktorze pozwalają na podanie nazwy pliku w formie łańcucha znaków (do takich klas należą między innymi `FileWriter` oraz `FileOutputStream`), pozwalają także na przekazanie obiektu `File`. Można stworzyć obiekt `File`, upewnić się, czy podana ścieżka jest poprawna (ewentualnie wykonać inne czynności), po czym użyć tego obiektu podczas tworzenia strumieni `FileWriter` lub `InputStream`.

Niektóre operacje, które można wykonać przy użyciu obiektów `File`:

- 1 Utworzenie obiektu `File` reprezentującego istniejący plik:

```
File p = new File("dane.txt");
```

- 2 Utworzenie nowego folderu:

```
File fld = new File("Rozdział7");
fld.mkdir();
```

- 3 Wyświetlenie zawartości folderu:

```
if (fld.isDirectory()) {
    String[] zawartoscFld = fld.list();
    for (int i = 0; i < zawartoscFld.length, i++) {
        System.out.println(zawartoscFld[i]);
    }
}
```

- 4 Pobranie bezwzględnej ścieżki do pliku lub folderu:

```
System.out.println(fld.getAbsolutePath());
```

- 5 Usunięcie pliku lub folderu
(w przypadku powodzenia metoda zwraca wartość true):

```
boolean czyUsunieto = p.delete();
```

Obiekt `File` reprezentuje nazwę i ścieżkę dostępu do pliku przechowywanego na dysku, na przykład:

`C:\Users\Kasia\Dane\gra.txt`

Jednak obiekt ten ani nie reprezentuje, ani nie zapewnia dostępu do faktycznej zawartości pliku!



Adres to NIE to samo co faktyczny budynek. Obiekt `File` można porównać z adresem — reprezentuje on nazwę i położenie pliku, lecz nie jego faktyczną zawartość

Obiekt `File` reprezentuje nazwę pliku „PlikGry.txt”.

`PlikGry.txt`

```
50, Elf, łuk, miecz, pył
200, Trol, pięści, wielki topór
120, Magik, czary, niewidzialność
```

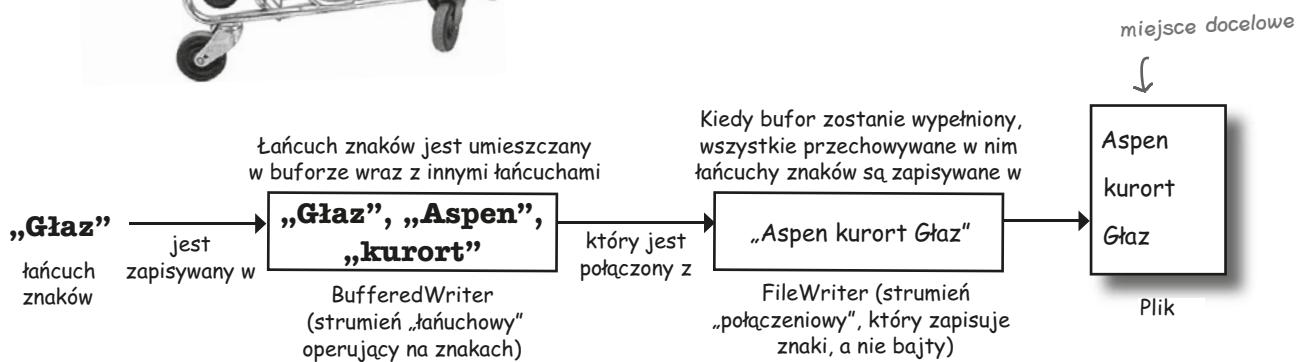
↑
Obiekt `File` nie reprezentuje (ani nie daje nam bezpośredniego dostępu) danych przechowywanych we wnętrzu pliku!

Piękno buforów

Operacje wejścia-wyjścia bez buforów można by porównać z robieniem zakupów bez koszyków lub wózków na towary. Każdą kupowaną rzecz trzeba by zanosić do samochodu niezależnie od pozostałych — jedną torbę zupy w proszku, jedną rolkę papieru toaletowego...



Bufory stanowią miejsce, w którym można tymczasowo gromadzić dane aż do momentu wypełnienia bufora (podobnie jak wózka na zakupy). W przypadku wykorzystania bufora trzeba będzie wykonać znacznie mniej operacji zapisu.



```
BufferedWriter pisarz = new BufferedWriter(new FileWriter(plik));
```

Ogromną zaletą buforów jest to, że ich stosowanie *ogromnie* poprawia efektywność wykonywanych operacji wejścia-wyjścia. Można zapisywać łańcuchy znaków w pliku wykorzystując jedynie strumień `FileWriter` oraz jego metodę `write(łańcuch)`, jednak w takim przypadku każde wywołanie metody będzie odpowiadało wykonaniu jednej operacji zapisu. Rozwiążanie takie przysparza narzutów, które nie są ani pożądane, ani konieczne, gdyż każda operacja na dysku to „poważna sprawa” w porównaniu z operacjami na danych wykonywanymi w pamięci. Dzięki połączeniu strumieni `BufferedWriter` oraz `FileWriter` pierwszy z nich będzie gromadzić wszystkie zapisywane dane aż do momentu wypełnienia bufora. Dopiero po wypełnieniu bufora strumień `FileWriter` zostanie poproszony o zapisanie danych w pliku na dysku.

Jeśli chcesz zapisać dane przed wypełnieniem bufora, możesz to zrobić. *Wystarczy opróżnić bufor*. W tym celu należy wywołać metodę `pisarz.flush()`, która nakazuje „wyślij do strumienia wszystko, co aktualnie znajduje się w buforze”.

Zwrócić uwagę, że nawet nie musimy przechowywać odwołania do obiektu `FileWriter`. Obchodzi nas wyłącznie obiekt `BufferedWriter`, gdyż to właśnie jego metody będziemy wywoływać, a gdy go zamknijemy, wykona za nas wszystkie niezbędne czynności.

Odczyt zawartości pliku tekstowego

Odczytanie zawartości pliku tekstowego jest proste. W poniższym przykładzie użyjemy obiektu `File`, który będzie reprezentować plik, obiektu `FileReader`, który wykona operację odczytu, oraz obiektu `BufferedReader`, który poprawi efektywność tej operacji.

Zawartość pliku będzie odczytywana wierszami wewnątrz pętli `while`, której wykonywanie zakończy się w momencie, gdy metoda `readLine()` zwróci wartość `null`. To najczęściej stosowany sposób odczytu danych (niemal każdego rodzaju, oprócz serializowanych obiektów): odczytuj dane w pętli `while` (a właściwie wewnątrz jej *testu*) i zakończ pętlę, kiedy nie pozostanie już nic, co można by odczytać (o tym poinformuje nas wartość `null` zwrócona przez dowolną używaną metodę odczytującą).

```
import java.io.*; Nie zapomnij o importie.
public class OdczytPliku {
    public static void main(String[] args) {
        try {
            File mojPlik = new File("tekst.txt");
            FileReader czytelnikF = new FileReader(mojPlik);
            BufferedReader czytelnik = new BufferedReader(czytelnikF);

Utwórz zmienną klasy String, w której będą przechowywane odczytywane wiersze tekstu.
            String wiersz = null;

            while ((wiersz = czytelnik.readLine()) != null) {
                System.out.println(wiersz);
            }
            czytelnik.close();
        } catch (Exception zd) {
            zd.printStackTrace();
        }
    }
}
```

FileReader to strumień nawiązujący połączenie z plikiem tekstowym i umożliwiający odczytywanie znaków.

Aby poprawić efektywność odczytu, należy połączyć strumień `FileReader` ze strumieniem `BufferedReader`. W tym przypadku dane będą pobierane z pliku dopiero wtedy, kiedy cały bufor zostanie opróżniony (co nastąpi, gdy program odczyta całą jego zawartość).

Oto co oznacza kod pętli: „Odczytaj wiersz tekstu i zapisz go w zmiennej »wiersz«. Tak dugo, jak długo wartość tej zmiennej jest różna od null (czyli COŚ udało się odczytać), wyświetlaj odczytany tańcuch znaków”.

To samo można wyrazić także w inny sposób: „Dopóki można odczytać jakieś wiersze tekstu, odczytuj je i wyświetlaj”.

Plik zawierający dwa wiersze tekstu.

Ile jest $2+2?/4$
Ile jest $20+22?/42$

tekst.txt

Kwiz — gra (zarys kodu)

```

public class KwizGra {

    public void doRoboty() {
        // utworzenie i wyświetlenie graficznego interfejsu użytkownika
    }

    public class NastepnaKartaListener implements ActionListener {
        public void actionPerformed(ActionEvent zd) {
            // jeśli aktualnie prezentowane jest pytanie, pokazujemy odpowiedź,
            // w przeciwnym razie pokazujemy następne pytanie
            // ustawienie flagi określającej, co jest prezentowane
        }
    }

    public class OtworzMenuItemListener implements ActionListener {
        public void actionPerformed(ActionEvent zd) {
            // wyświetlenie okna dialogowego obsługującego plików
            // umożliwienie użytkownikowi wybrania pliku, który chce otworzyć
        }
    }

    private void wczytajPlik(File plik) {
        // należy stworzyć tablicę ArrayList zawierającą karty odczytując dane z pliku
        // metoda jest wywoływana przez klasę OtworzMenuItemListener, odczytuje zawartość
        // pliku wiersz po wierszu i dla każdego z nich wywołuje metodę tworzMete(), aby
        // utworzyć kartę
        // (każdy wiersz w pliku zawiera zarówno pytanie, jak i odpowiedź,
        // oddzielone od siebie znakiem "/")
    }

    private void tworzMete(String wierszDanych) {
        // wywoływana przez metodę wczytajPlik(), pobiera wiersze danych z pliku tekstowego
        // i dzieli na dwie części – pytanie i odpowiedź – na podstawie których tworzy obiekt
        // kartaKwizowa i dodaje go do tablicy kart (tablicaKart) typu ArrayList
    }
}

```

Kod klasy KwizGra

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class KwizGra {

    private JTextArea pytanie;
    private JTextArea odpowiedz;
    private ArrayList<KartaKwizowa> listaKart;
    private KartaKwizowa biezacaKarta;
    private int indeksBiezacejKarty;
    private JFrame ramka;
    private JButton przyciskNastepnaKarta;
    private boolean czyPrezentowanaOdpowiedz;

    public static void main (String[] args) {
        KwizGra gra = new KwizGra();
        gra.dodziela();
    }

    public void dodziela() {

        // tworzymy graficzny interfejs użytkownika

        ramka = new JFrame("Kwiz");
        ramka.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panelGlowny = new JPanel();
        Font czcionkaDuza = new Font("sanserif", Font.BOLD, 24);

        pytanie = new JTextArea(10,20);
        pytanie.setFont(czcionkaDuza);
        pytanie.setLineWrap(true);
        pytanie.setEditable(false);

        JScrollPane przewijanieP = new JScrollPane(pytanie);
        przewijanieP.setVerticalScrollBarPolicy(
            ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        przewijanieP.setHorizontalScrollBarPolicy(
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
        przyciskNastepnaKarta = new JButton("Pokaż pytanie");
        panelGlowny.add(przewijanieP);
        panelGlowny.add(przyciskNastepnaKarta);
        przyciskNastepnaKarta.addActionListener(new NastepnaKartaListener());

        JMenuBar pasekMenu = new JMenuBar();
        JMenu menuPlik = new JMenu("Plik");
        JMenuItem opcjaOtworz = new JMenuItem("Otwórz zbiór kart");
        opcjaOtworz.addActionListener(new OtworzMenuListener());
        menuPlik.add(opcjaOtworz);
        pasekMenu.add(menuPlik);
        ramka.setJMenuBar(pasekMenu);
        ramka.getContentPane().add(BorderLayout.CENTER, panelGlowny);
        ramka.setSize(640,500);
        ramka.setVisible(true);
    } // koniec metody
}
```

Na tej stronie przedstawiony jest wyłącznie kod odpowiedzialny za stworzenie graficznego interfejsu użytkownika — nic specjalnego!

```

public class NastepnaKartaListener implements ActionListener {
    public void actionPerformed(ActionEvent zd) {
        if (czyPrezentowanaOdpowiedz) {
            // pokaż odpowiedź, bo użytkownik już widział pytanie
            pytanie.setText(biezacaKarta.getOdpowiedz());
            przyciskNastepnaKarta.setText("Następna karta");
            czyPrezentowanaOdpowiedz = false;
        } else {
            // pokaż następne pytanie
            if (indeksBiezacejKarty < listaKart.size()) {
                pokazNastepnaKarte();
            } else {
                // nie ma więcej kart
                pytanie.setText("To była ostatnia karta");
                przyciskNastepnaKarta.setEnabled(false);
            }
        }
    }
}

public class OtworzMenuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
        JFileChooser dialogFile = new JFileChooser();
        dialogFile.showOpenDialog(ramka);
        wczytajPlik(dialogFile.getSelectedFile());
    }
}

private void wczytajPlik(File file) {
    listaKart = new ArrayList<KartaKwizowa>();
    try {
        BufferedReader czytelnik = new BufferedReader(new FileReader(file));
        String wiersz = null;
        while ((wiersz = czytelnik.readLine()) != null) {
            tworzMatek(wiersz);
        }
        czytelnik.close();
    } catch (Exception ex) {
        System.out.println("Nie można odczytać pliku z kartami!");
        ex.printStackTrace();
    }
    // czas zaczynać
    pokazNastepnaKarte();
}

private void tworzMatek(String wierszDanych) {
    String[] wyniki = wierszDanych.split("/");
    KartaKwizowa card = new KartaKwizowa(wyniki[0], wyniki[1]);
    listaKart.add(card);
    System.out.println("utworzno kartę");
}

private void pokazNastepnaKarte() {
    biezacaKarta = listaKart.get(indeksBiezacejKarty);
    indeksBiezacejKarty++;
    pytanie.setText(biezacaKarta.getPytanie());
    przyciskNastepnaKarta.setText("Pokaż odpowiedź");
    czyPrezentowanaOdpowiedz = true;
}
} // koniec klasy

```

Sprawdzamy flagę logiczną `czyPrezentowanaOdpowiedz`, aby sprawdzić, czy użytkownik aktualnie ogląda odpowiedź czy pytanie, oraz aby wykonać odpowiednie czynności w zależności od uzyskanej odpowiedzi.

Wyświetlamy okno dialogowe do obsługi plików i dajemy użytkownikowi możliwość wybrania i otwarcia pliku.

Tworzymy strumień `BufferedReader` połączony ze strumieniem `FileReader`, a do drugiego z nich przekazujemy obiekt `file` reprezentujący plik wybrany przez użytkownika w oknie dialogowym.

Odczytujemy zawartość pliku wiersz po wierszu, przekazując odczytane wiersze do metody `tworzMatek()`, która je przetwarza i na ich podstawie tworzy obiekty `KartaKwizowa`, które są następnie dodawane do obiektu `ArrayList`.

Każdy wiersz kodu odpowiada jednej kartecce kwizowej, jednak musimy przetworzyć je w taki sposób, aby uzyskać pytanie i odpowiedź reprezentowane przez niezależne ciągi znaków. W tym celu wykorzystujemy metodę `split()`. Zagadnienie to opiszymy na następnej stronie.

Przetwarzaniełańcuchów przy użyciu metody `split()`

Przetwarzaniełańcuchów znaków przy użyciu metody `split()`

Wyobraź sobie, że masz następującą karteczkę kwizową:



Zapisaną w pliku tekstowym w następujący sposób:

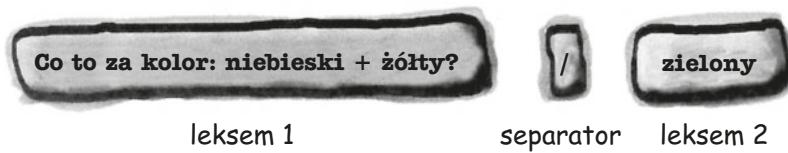
Co to za kolor: niebieski + żółty/zielony
Co to za kolor: czerwony + niebieski/purpurowy

W jaki sposób można oddzielić pytanie od odpowiedzi?

Podczas odczytywania pliku pytanie i odpowiedź są połączone ze sobą i zapisane w jednym wierszu, oddziela je jedynie znak ukośnika „/” (właśnie w taki sposób zapisujemy zawartość pliku).

Metoda `split()` klasy `String` pozwala podzielićłańcuch znaków na części

Obiekt tej klasy mówi: „przekaż mi obiekt `String` i znak stanowiący separator, a ja podzielię tenłańcuch na części”.



```
String[] testowy = "Co to za kolor: niebieski + żółty/zielony";
```

```
String[] wyniki = testowy.split("/");  
for (String leksem:wyniki) {  
    System.out.println(leksem);  
}
```

Przeglądamy tablicę w pętli i wyświetlamy każdy leksem (fragment). W tym przypadku istnieją tylko dwa leksem: "Co to za kolor: niebieski + żółty" oraz "zielony".

Metoda `split()` otrzymujełańcuch "/" i używa go do podzieleniałańcucha znaków na (w naszym przypadku) dwie części.

(Uwaga: metoda `split()` ma ZNACZNIE większe możliwości od tych, które wykorzystujemy w tym przykładzie. Może dokonywać niezwykle złożonej analizyłańcuchów znaków, z wykorzystaniem filtrów, znaków wieloznacznych itd.).

Nie istniejąca głupie pytania

P: No dobrze, zatrzałem do dokumentacji API i okazało się, że w pakiecie `java.io` jest około pięciu milionów różnych klas. Jakim cudem mam się zorientować, której z nich użyć?

O: Interfejs programistyczny służący do obsługi operacji wejścia-wyjścia wykorzystuje pojęcie „modularnego” łączenia strumieni, dzięki czemu możesz łączyć ze sobą strumienie „połączeniowe” i „łańcuchowe” (nazywane także „filtrami”), tworząc szeroką gamę kombinacji, mogących zaspokajać wszystkie Twoje potrzeby.

Takie „łańcuchy” nie muszą składać się tylko z dwóch strumieni — możesz łączyć dowolnie dużo strumieni „łańcuchowych”, tworząc kombinacje, która zapewni Ci niezbędne możliwości funkcjonalne.

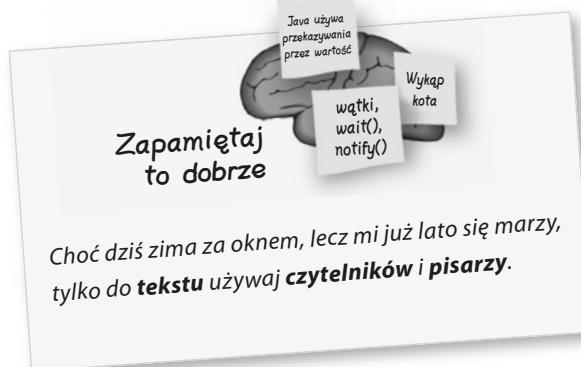
Niemniej jednak w większości przypadków będziesz stosować tę samą, niewielką grupę klas. W przypadku operacji na plikach tekstowych prawdopodobnie w zupełności wystarczą Ci strumienie `BufferedReader` oraz `BufferedWriter` (połączone ze strumieniami `FileReader` oraz `FileWriter`). W razie operowania na serializowanych obiektach możesz posługiwać się strumieniami `ObjectOutputStream` i `ObjectInputStream` (połączone ze strumieniami `FileOutputStream` oraz `FileInputStream`).

A zatem około 90 procent operacji wejścia-wyjścia, jakie zazwyczaj możesz wykonywać, wymaga użycia klas, które już poznajeś.

P: A co z nową biblioteką do obsługi wejścia-wyjścia — określana jako „nio” — dodaną w Javie 1.4?

O: Klasy należące do pakietu `java.nio` stanowią znaczny postęp, jeśli chodzi o efektywność działania, oraz w zdecydowanie większym stopniu (i lepiej) wykorzystują możliwości systemu i komputera, na którym jest uruchamiany program. Jedną z podstawowych cech „nio” jest możliwość sprawowania bezpośredniej kontroli na buforami. Kolejną dodaną możliwością są „nieblokujące” operacje wejścia-wyjścia, co oznacza, że kod nie musi już czekać, jeśli nie ma już niczego, co można by odczytać lub zapisać. Niektóre z istniejących klas (w tym także `FileInputStream` oraz `FileOutputStream`) w niewidoczny sposób korzystają z tych nowych możliwości. Jednak korzystanie z klas NIO jest nieco trudniejsze, zatem jeśli nie masz korzystać z nowych możliwości, możesz dojść do wniosku, że lepsze będzie użycie prostszych rozwiązań przedstawionych w tej książce. Co więcej, jeśli klasy NIO nie będą stosowane uważnie, mogą spowodować obniżenie efektywności działania programu. Można sądzić, że w 90 procentach standardowych zastosowań starsze rozwiązania będą całkowicie wystarczające, zwłaszcza jeśli dopiero zaczynasz uczyć się Javy.

Niemniej jednak możesz sobie ułatwić wykorzystanie klas NIO, używając strumienia `FileInputStream` i pobierając jego kanał przy użyciu metody `getChannel()` (dodanej do klasy `FileInputStream` w wersji 1.4 Javy).



Choć dziś zima za oknem, lecz mi już lato się marzy,
tylko do tekstu używaj czytelników i pisarzy.

CELNE SPOSTRZEŻENIA

- Aby zapisać plik tekstowy, zacznij od utworzenia strumienia `FileWriter`.
- Następnie, aby poprawić efektywność operacji, połącz strumień `FileWriter` ze strumieniem `BufferedWriter`.
- Obiekt `File` reprezentuje plik znajdujący się w określonym miejscu, nie reprezentuje jednak zawartości tego pliku.
- Posługując się obiektem `File`, można odczytywać zawartość folderów, poruszać się po nich oraz tworzyć lub usuwać.
- Większość strumieni wymagających podania nazwy pliku zapisanych w formie łańcucha znaków, pozwala także na przekazywanie obiektów `File`, a stosowanie tych obiektów jest bezpieczniejsze.
- Aby odczytać zawartość pliku tekstowego, zacznij od utworzenia strumienia `FileReader`.
- Aby poprawić efektywność odczytu, połącz strumień `FileReader` ze strumieniem `BufferedReader`.
- Aby przeanalizować zawartość pliku tekstowego, musisz mieć pewność, że podczas jej zapisywania były używane jakieś separatory, które pozwolą wyróżnić poszczególne elementy. Często spotykanym rozwiązaniem jest zastosowanie wybranego znaku, oddzielającego poszczególne elementy zawartości.
- Aby podzielić tekst na leksemy, można skorzystać z metody `split()` klasy `String`. Łącuch znaków, w którym umieszczone jeden separator, będzie zawierać dwa leksem, po jednym z każdej strony separatora. *Sam separator nie jest traktowany jak leksem.*

Identyfikator wersji — wielki problem serializacji

Przekonaleś się już, że operacje wejścia-wyjścia w Javie są całkiem proste, zwłaszcza jeśli ograniczysz się do najczęściej stosowanych kombinacji strumieni. Jednak istnieje pewne zagadnienie, które może być *naprawdę* ważne.

Kontrola wersji obiektów ma kluczowe znaczenie!

Jeśli korzystasz z mechanizmu serializacji, musisz mieć dostęp do klasy, abyś mógł odtworzyć zapisany obiekt i użyć go w programie. No tak, to nie podlega dyskusji. Jest jednak pewien problem, który nie jest już tak oczywisty. Otóż: co się dzieje, jeśli została *zmieniona klasa*? Ups. Wyobraź sobie próbę deserializacji obiektu Pies w sytuacji, gdy typ jednej z jego składowych (i to tych zapamiętywanych) zmienił się z double na String. Taki przypadek w bardzo poważny sposób naruszyłby zasady bezpieczeństwa typów stosowane w Javie. Niemniej jednak nie jest to jedyna sytuacja, która może zaalarmować mechanizmy zabezpieczeń języka. Zastanów się nad kilkoma przypadkami.

Zmiany, które mogą uniemożliwić przeprowadzenie deserializacji:

Usunięcie składowej.

Zmiana zadeklarowanego typu składowej.

Oznaczenie składowej, która wcześniej była zapisywana, słowem kluczowym transient.

Przesunięcie klasy w górę lub ku dołowi hierarchii dziedziczenia.

Zaimplementowanie interfejsu Serializable lub usunięcie tego interfejsu w którejś z klas należących do grafu obiektu.

Zmiana składowej z „normalnej” na statyczną.

Zmiany, które zazwyczaj nie przysparzają problemów:

Dodanie do klasy nowych składowych (istniejące obiekty zostaną odtworzone, a nowe składowe, których nie było w momencie przeprowadzania serializacji, uzyskają wartości domyślne).

Dodanie klas do drzewa dziedziczenia.

Usunięcie klas z drzewa dziedziczenia.

Zmiana poziomu dostępu do składowych nie ma żadnego wpływu na zapisywanie wartości w składowych podczas procesu deserializacji.

Uwzględnienie w procesie serializacji składowej, która wcześniej była pomijana (podczas odtwarzania obiektów, które zostały serializowane wcześniej, takie składowe przyjmą wartości domyślne).

1 Tworzyesz klasę Pies

101101 1011
1010100010
1010 10 0
01010 1
1010101
10101010
1001010101 1 0
1 10 10

identyfikator wersji klasy #343

Pies.class

2 Serializujesz obiekt Pies, wykorzystując istniejącą klasę.

Obiekt Pies
Obiekt jest oznaczony numerem wersji #343

3 Zmieniasz klasę Pies

101101 1011
10100010 1010
10 0 01010 1
100001 1010 0
00110101 1 0
1 10 10

identyfikator wersji klasy #728

Pies.class

4 Próbujesz przeprowadzić deserializację obiektu, wykorzystując przy tym zmienioną klasę.

Obiekt Pies
Obiekt jest oznaczony numerem wersji #343
Pies.class

identyfikator wersji klasy #728

5 Próba deserializacji kończy się niepowodzeniem.

Wirtualna maszyna Javy stwierdza: „Nie możesz nauczyć starego obiektu Pies wykorzystania nowego kodu”.

Stosowanie serialVersionUID

Z każdym razem, gdy obiekt jest zapisywany w procesie serializacji, zostaje on (jak również wszystkie inne obiekty znajdujące się w jego grafie) „opatrzony” identyfikatorem wersji aktualnej klasy tego obiektu. Identyfikator ten nosi nazwę serialVersionUID i jest wyznaczany na podstawie informacji o strukturze klasy. Jeśli od czasu zapisania obiektu jego klasa została zmodyfikowana, to podczas próby jego deserializacji wartość serialVersionUID może być inna — w takim przypadku nie uda się poprawnie zakończyć deserializacji! Niemniej jednak możesz nad tym zapanować.

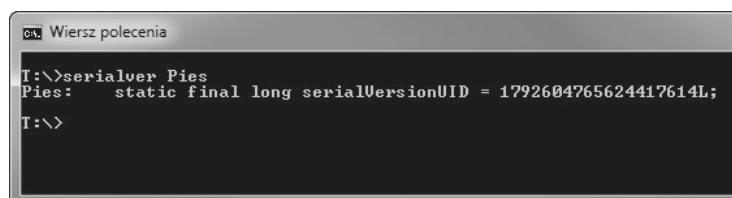
Jeśli podejrzewasz, że istnieje jakiekolwiek, nawet najmniejsze, prawdopodobieństwo zmodyfikowania klasy, umieść w jej kodzie identyfikator wersji.

Kiedy Java stara się odtworzyć serializowany obiekt, porównuje jego wartość serialVersionUID z analogiczną wartością klasy używanej przez JVM w procesie deserializacji. Na przykład, jeśli serializowany obiekt Pies ma identyfikator wersji o numerze, dajmy na to, 23 (w rzeczywistości wartości serialVersionUID są znacznie wyższe), to podczas przeprowadzania jego deserializacji Java najpierw porówna wartość serialVersionUID obiektu z wartością serialVersionUID klasy. Jeśli obie te wartości nie będą identyczne, Java uzna, że dostępna klasa nie jest zgodna z zapisanym obiektem, a próba deserializacji zakończy się zgłoszeniem wyjątku.

A zatem rozwiążanie tego problemu polega na podaniu w klasie wartości serialVersionUID. Dzięki temu, gdy w przyszłości klasa będzie się zmieniać, wartość serialVersionUID pozostanie taka sama, a wirtualna maszyna Javy stwierdzi: „w porządku, super, klasa jest zgodna z serializowanym obiektem”, nawet jeśli klasa w rzeczywistości została zmieniona.

Rozwiążanie to będzie działać prawidłowo *wyłącznie* w przypadkach, gdy zmiany w klasie będziesz wprowadzać bardzo ostrożnie! Innymi słowy, teraz to *Ty* bierzesz na siebie odpowiedzialność za wszystko, co może się wydarzyć podczas próby deserializacji starego obiektu przy wykorzystaniu nowej klasy.

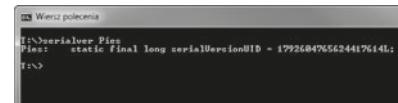
Aby określić wartość serialVersionUID dla klasy, użij programu narzędziowego serialver dostarczanego wraz z JDK.



```
Wiersz polecenia
T:\>serialver Pies
Pies: static final long serialVersionUID = 1792604765624417614L;
T:\>
```

Oto co powinieneś zrobić, jeśli przypuszczasz, że po przeprowadzeniu serializacji obiektów tej klasy, jej struktura może się zmienić:

- Użij programu serialver, aby określić identyfikator wersji klasy.



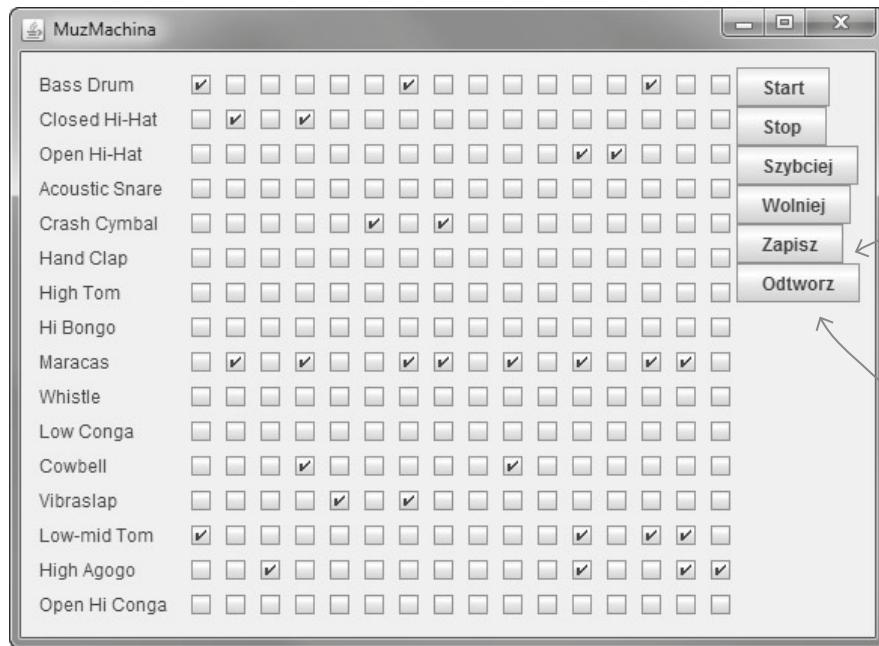
```
Wiersz polecenia
T:\>serialver Pies
Pies: static final long serialVersionUID = 1792604765624417614L;
T:\>
```

- Skopiuj wygenerowane wyniki i umieść je w kodzie klasy.

```
public class Pies {
    static final int long serialVersionUID
        = 1792604765624417614L;
    private String imie;
    private int wielosc;
    // dalszy kod klasy
}
```

- Pamiętaj, że jeśli modyfikujesz klasę, bierzesz na siebie odpowiedzialność za wprowadzone zmiany! Na przykład upewnij się, że w sytuacji deserializacji starych obiektów Pies, nowa klasa Pies jest w stanie poradzić sobie z domyślnymi wartościami składowych dodanych po zapisaniu obiektu.

Kod od kuchni



Kiedy klikniesz przycisk „Zapisz”, bieżąca kompozycja zostanie zapisana w pliku.

Kliknięcie przycisku „Odtwarz” spowoduje odtworzenie zapisanej kompozycji i odpowiednie zaznaczenie pól wyboru.

Niech nasza muzyczna aplikacja będzie w stanie zapisywać i odczytywać nasze ulubione kompozycje.

Zapisywanie kompozycji

Pamiętasz zapewne, że w naszej muzycznej aplikacji kompozycja nie jest niczym więcej jak tylko grupą zaznaczonych pól wyboru. Kiedy trzeba ją odtworzyć, program przegląda wszystkie pola wyboru i określa, jakie dźwięki mają być odtwarzane w każdym z szesnastu taktów. A zatem, zapisanie kompozycji sprowadza się do zapisania stanu pól wyboru.

Możemy stworzyć prostą tablicę wartości logicznych przechowującą stan wszystkich 256 pól wyboru. Obiekt tablicy można serializować, o ile tylko jej *zawartość* na to pozwala. Zatem nie będziemy mieć najmniejszych problemów z zapisaniem tablicy wartości logicznych.

Aby odczytać utwór, odczytujemy jeden obiekt tablicy (odtwarzamy go) i przywracamy odpowiednie stany wszystkich pól wyboru. Większość kodu naszej aplikacji widziałeś już wcześniej w „Kodzie od kuchni”, w którym stworzyliśmy jej interfejs graficzny. Dlatego też w tej części rozdziału przedstawimy tylko te fragmenty kodu, które są odpowiedzialne za zapisywanie oraz odtwarzanie kompozycji.

Informacje przedstawione w tym „Kodzie od kuchni” przygotują nas do zagadnień, jakimi będziemy się zajmować w następnym rozdziale, w którym kompozycja będzie zapisywana nie w *pliku*, lecz na serwerze, za pośrednictwem *sieci*. Poza tym, zamiast *wczytywania* kompozycji z pliku, będziemy je odbierać z *serwera* za każdym razem, gdy jakaś inna osoba zapisze je na serwerze.

Serializacja kompozycji

To wewnętrzna klasa umieszczona w kodzie naszej aplikacji.

```
public class ZapiszListener implements ActionListener {
    public void actionPerformed(ActionEvent e) { ←
        // tworzymy tablicę ze stranami pól wyboru
        boolean[] stanyPol = new boolean[256]; ←
        for (int i = 0; i < 256; i++) {
            JCheckBox pole = (JCheckBox) listaPolWyboru.get(i);
            if (pole.isSelected()) {
                stanyPol[i] = true;
            }
        } ←
        try {
            FileOutputStream strumienPlk = new FileOutputStream(new File("kompozycja.ser"));
            ObjectOutputStream os = new ObjectOutputStream(strumienPlk);
            os.writeObject(stanyPol);
        } catch (Exception ex) {
            ex.printStackTrace();
        } ←
    } // koniec metody
} // koniec klasy
```

Wszystkie te operacje zostają wykonane, gdy użytkownik kliknie przycisk i zostanie wygenerowane zdarzenie ActionEvent.

Tworzymy tablicę wartości logicznych przechowującą stany wszystkich pól wyboru.

Przeglądamy listę pól wyboru (typu ArrayList), odczytujemy stan każdego z pól i zapisujemy go w tablicy typu boolean.

To część kodu jest bardzo prosta. Po prostu zapisujemy (serializujemy) tablicę wartości logicznych.

Odtwarzanie zapisanej kompozycji

To mniej więcej te same operacje, tylko „w przeciwnym kierunku” — w tym przypadku odczytujemy tablicę wartości logicznych i na jej podstawie odtwarzamy stan pól wyboru wchodzących w skład interfejsu użytkownika naszej aplikacji. Wszystkie te operacje są wykonywane, gdy użytkownik kliknie przycisk *Odtworz*.

Odtwarzanie kompozycji

```
public class OdtworzListener implements ActionListener {  
    public void actionPerformed(ActionEvent a) {  
  
        boolean[] stanyPol = null;  
        try {  
            FileInputStream plikDanych = new FileInputStream(new File("kompozycja.ser"));  
            ObjectInputStream is = new ObjectInputStream(plikDanych);  
            stanyPol = (boolean[]) is.readObject(); ←  
        } catch(Exception ex) {ex.printStackTrace();}  
  
        for (int i = 0; i < 256; i++) {  
            JCheckBox pole = (JCheckBox) listaPolWyboru.get(i);  
            if (stanyPol[i]) {  
                pole.setSelected(true);  
            } else {  
                pole.setSelected(false);  
            }  
        }  
  
        sekvenser.stop();  
        utworzSciezkeIOdtworz();  
    } // koniec metody  
} // koniec klasy wewnętrznej
```

To kolejna klasa wewnętrzna umieszczona w kodzie naszej aplikacji.

Odczytujemy z pliku jeden obiekt (tablicę typu `boolean`) i rzutujemy na właściwy typ (pamiętaj, że metoda `readObject()` zwraca odwołanie typu `Object`).

Teraz odtwarzamy stany wszystkich pól wyboru (`JCheckBox`) umieszczonych na liście pól wyboru (`ArrayList`).

Zatrzymujemy jakikolwiek utwór, który sekvenser aktualnie odtwarza, i konstruujemy nową sekwencję na podstawie bieżących stanów pól wyboru przechowywanych w `ArrayList`.

Zaostrz ołówek

Ta wersja programu ma jedno bardzo poważne ograniczenie! Po kliknięciu przycisku *Zapisz kompozycję* jest zapisywana w pliku o nazwie *kompozycja.ser* (który zostaje utworzony, jeśli wcześniej nie istniał). Jednak za każdym razem, gdy zapiszesz kompozycję, poprzednia zostanie utracona.

Usprawnij opcje zapisu i odtwarzania, wykorzystując obiekt `JFileChooser`, aby dać użytkownikowi możliwość wyboru nazwy pliku oraz zapisywania i odtwarzania dowolnej ilości kompozycji.



Czy można je zapisać?

Jak myślisz, które z poniższych obiektów można lub powinno się dać zapisywać przy wykorzystaniu mechanizmu serializacji? Jeśli nie można tego zrobić, to dlaczego? Po odtworzeniu obiekt nie miałby to sensu? A może z powodów bezpieczeństwa? Może obiekt jest w stanie działać tylko podczas jednego uruchomienia wirtualnej maszyny Javy? Postaraj się zgadnąć bez zaglądania do dokumentacji Javy.

Typ obiektu	Zdatny do serializacji?	Jeśli nie, to dlaczego?
Object	Tak/Nie	_____
String	Tak/Nie	_____
File	Tak/Nie	_____
Date	Tak/Nie	_____
OutputStream	Tak/Nie	_____
JFrame	Tak/Nie	_____
Integer	Tak/Nie	_____
System	Tak/Nie	_____

Co jest dozwolone?

Zaznacz fragmenty kodu, które można skompilować (zakładając, że zostaną umieszczone w poprawnie napisanej klasie).



```
FileReader czytelnikF = new FileReader();
BufferedReader czytelnik = new BufferedReader(czytelnikF);

FileOutputStream f = new FileOutputStream(new File("test.ser"));
ObjectOutputStream os = new ObjectOutputStream(f);

BufferedReader czytelnik = new BufferedReader(new FileReader(plik));
String wiersz = null;
while ((wiersz = czytelnik.readLine()) != null) {
    tworzKarte(wiersz);
}

ObjectInputStream is = ObjectInputStream(new FileInputStream("gra.ser"));
Bohater postac1 = (Bohater) is.readObject();
```



W tym rozdziale przedstawiony został wspaniały świat operacji wejścia-wyjścia w Javie. Twoim zadaniem jest określenie, czy przedstawione poniżej stwierdzenia są prawdziwe czy nie.

👉 PRAWDA CZY FAŁSZ 👈

1. W przypadku zapisywania danych, które będą wykorzystywane nie tylko przez programy napisane w Javie, serializacja jest poprawnym rozwiązaniem.
2. Stan obiektu można zapisać wyłącznie przy wykorzystaniu mechanizmu serializacji.
3. Do zapisywania serializowanych obiektów jest używana klasa `ObjectOutputStream`.
4. Strumienie „łańcuchowe” mogą być używane samodzielnie bądź też wraz ze strumieniami „połączonymi”.
5. Jedno wywołanie metody `writeObject()` może spowodować zapisanie wielu obiektów.
6. Domyślnie wszystkie klasy można serializować.
7. Modyfikator `transient` pozwala na oznaczanie składowych, które mają być uwzględniane w procesie serializacji.
8. Jeśli nie można przeprowadzić serializacji klasy bazowej, to nie będzie to także możliwe dla klasy potomnej.
9. Podczas przeprowadzania deserializacji obiektów są one odtwarzane w odwrotnej kolejności, czyli ten, który został zapisany jako ostatni, będzie odtworzony jako pierwszy.
10. Podczas przeprowadzania deserializacji obiektu nie jest wykonywany jego konstruktor.
11. Zarówno podczas serializacji, jak i zapisu danych do zwykłego pliku tekstowego mogą być zgłaszane wyjątki.
12. Strumień `BufferedWriter` można połączyć ze strumieniem `FileWriter`.
13. Obiekty `File` reprezentują pliki, lecz nie foldery.
14. Nie można zmusić bufora do zapisania umieszczonych w nim danych aż do momentu, gdy bufor zostanie wypełniony w całości.
15. Zarówno podczas zapisu, jak i odczytu plików tekstowych można wykorzystywać bufory.
16. W przypadku korzystania z metody `split()` klasy `String` separator jest traktowany jako leksem i dołączany do tablicy wynikowej.
17. *Jakakolwiek* modyfikacja klasy sprawia, że nie będzie można przeprowadzić deserializacji zapisanych wcześniej obiektów tej klasy.



Magnesiki z kodem

To zadanie jest trudne, zatem nadaliśmy mu wyższą rangę, zmieniając z „ćwiczenia” na „zagadkę”. Wykorzystując przedstawione magnesiki z kodem, zrekonstruuj działający program generujący poniższe wyniki. (Być może nie będziesz musiał używać wszystkich magnesików, może się także zdarzyć, że niektórych magnesików będziesz musiał użyć kilka razy).

```

class GraPrzygodowa implements Serializable {
    try {
        FileOutputStream fos = new
        FileOutputStream("gra.ser");
        short getZ() {
            return z;
        }
        e.printStackTrace();
        oos.close();
    }
    int getX() {
        return x;
    }
    System.out.println(g.getX() + g.getY() + g.getZ());
}

public int x = 3;
transient long y = 4;
private short z = 5;

long getY() {
    return y;
}
class GraPrzygodowaTester {
    import java.io.*;
    } catch (Exception e) {
        g = (GraPrzygodowa) ois.readObject();
    }
    ois.close();
    fos.writeObject(g);
}

```

Wiersz polecenia

```
T:\>java GraPrzygotowaTester
12
8
```

```
ObjectOutputStream oos = new
ObjectOutputStream(fos);
```

```
oos.writeObject(g);
```

```
public static void main(String[] args) {
    GraPrzygodowa g = new GraPrzygodowa();
```



Rozwiązania ćwiczeń

1. W przypadku zapisywania danych, które będą wykorzystywane nie tylko przez programy napisane w Javie, serializacja jest poprawnym rozwiązaniem. **Fałsz**
2. Stan obiektu można zapisać wyłącznie przy wykorzystaniu mechanizmu serializacji. **Fałsz**
3. Do zapisywania serializowanych obiektów jest używana klasa `ObjectOutputStream`. **Prawda**
4. Strumienie „łańcuchowe” mogą być używane samodzielnie bądź też wraz ze strumieniami „połączonymi”. **Fałsz**
5. Jedno wywołanie metody `writeObject()` może spowodować zapisanie wielu obiektów. **Prawda**
6. Domyslnie wszystkie klasy można serializować. **Fałsz**
7. Modyfikator `transient` pozwala na oznaczanie składowych, które mają być uwzględniane w procesie serializacji. **Fałsz**
8. Jeśli nie można przeprowadzić serializacji klasy bazowej, to nie będzie to także możliwe dla klasy potomnej. **Fałsz**
9. Podczas przeprowadzania deserializacji obiektów są one odtwarzane w odwrotnej kolejności, czyli ten, który został zapisany jako ostatni, będzie odtworzony jako pierwszy. **Fałsz**
10. Podczas przeprowadzania deserializacji obiektu nie jest wykonywany jego konstruktor. **Prawda**
11. Zarówno podczas serializacji, jak i zapisu danych do zwykłego pliku tekstowego mogą być zgłaszane wyjątki. **Prawda**
12. Strumień `BufferedWriter` można połączyć ze strumieniem `FileWriter`. **Prawda**
13. Obiekty `File` reprezentują pliki, lecz nie foldery. **Fałsz**
14. Nie można zmusić bufora do zapisania umieszczonych w nim danych aż do momentu, gdy bufor zostanie wypełniony w całości. **Fałsz**
15. Zarówno podczas zapisu, jak i odczytu plików tekstowych można wykorzystywać bufory. **Prawda**
16. W przypadku korzystania z metody `split()` klasy `String` separator jest traktowany jako leksem i dołączany do tablicy wynikowej. **Fałsz**
17. *Jakakolwiek* modyfikacja klasy sprawia, że nie będzie można przeprowadzić deserializacji zapisanych wcześniej obiektów tej klasy. **Fałsz**



To dobrze, że w końcu doczekaliśmy się odpowiedzi. Ten rozdział zaczynał mnie już męczyć.



```
import java.io.*;  
  
class GraPrzygodowa implements Serializable {  
    public int x = 3;  
    transient long y = 4;  
    private short z = 5;  
  
    int getX() {  
        return x;  
    }  
  
    long getY() {  
        return y;  
    }  
  
    short getZ() {  
        return z;  
    }  
}  
  
class GraPrzygodowaTester {  
    public static void main(String[] args) {  
        GraPrzygodowa g = new GraPrzygodowa();  
        System.out.println(g.getX() + g.getY() + g.getZ());  
        try {  
            FileOutputStream fos = new FileOutputStream("gra.ser");  
            ObjectOutputStream oos = new ObjectOutputStream(fos);  
            oos.writeObject(g);  
            oos.close();  
            FileInputStream fis = new FileInputStream("gra.ser");  
            ObjectInputStream ois = new ObjectInputStream(fis);  
            g = (GraPrzygodowa) ois.readObject();  
            ois.close();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        System.out.println(g.getX() + g.getY() + g.getZ());  
    }  
}
```

```
ca. Wiersz polecenia  
T:>java GraPrzygotowaTester  
12  
8
```

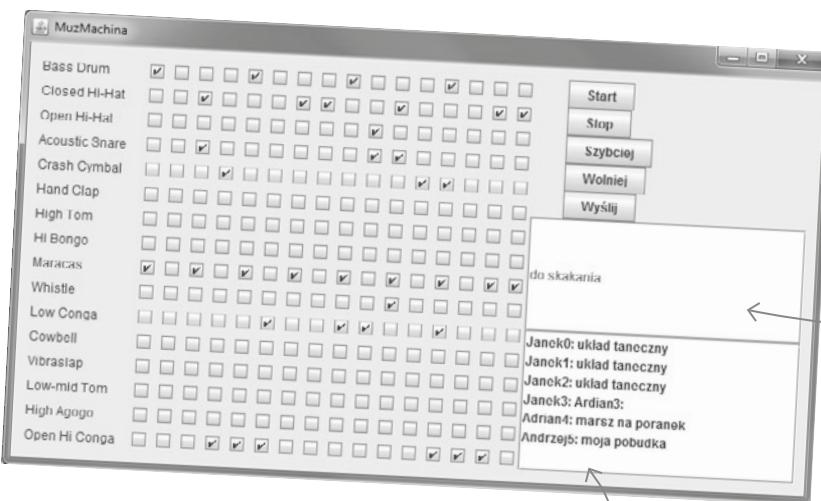

15. Zagadnienia sieciowe i wątki

Nawiąż połączenie



Nawiąż połączenie ze światem zewnętrznym. Twój program pisany w Javie może „wyciągnąć rękę i dotknąć” programu działającego na zupełnie innym komputerze. To całkiem łatwe. Wszystkie szczegóły niskiego poziomu związane z komunikacją sieciową są realizowane przez klasy należące do biblioteki java.net. Jedną z ogromnych zalet Javy jest to, że komunikacja sieciowa bardzo przypomina zwyczajną obsługę wejścia-wyjścia, a obie operacje różnią się jedynie obiektami umieszczanymi na samym końcu łańcucha strumieni. Jeśli dysponujesz strumieniem BufferedReader, to po prostu *odczytujesz* z niego znaki. Sam strumień BufferedReader nie zwraca żadnej uwagi na to, czy dane pochodzą z pliku czy też „spływają” kablem sieciowym. W tym rozdziale nawiążemy połączenie ze światem zewnętrznym, wykorzystując przy tym gniazda. Stworzymy gniazda *klienta*. Stworzymy także gniazda *serwera*. Będziemy pisać zarówno aplikacje będące *klientami*, jak i *serwery*. Sprawimy, że oba te rodzaje programów będą się ze sobą komunikować. Zanim ten rozdział się skończy, będziesz dysponować w pełni funkcjonalnym, wielowątkowym klientem do obsługi pogawędek internetowych. Czy użyliśmy słowa „wielowątkowy”? O tak! W tym rozdziale *zdobędziesz* tajemną wiedzę o tym, jak rozmawiać z Jurkiem i jednocześnie słuchać Zuzi.

Muzyczne pogawędkи w czasie rzeczywistym



Wpisz wiadomość i kliknij przycisk Wyślij, aby wystać zarówno tekst, **JAK RÓWNIEZ** bieżącą kompozycję.

Pracujesz nad stworzeniem gry komputerowej. Twój zespół zajmuje się opracowaniem podkładu muzycznego dla każdej z części gry. Dzięki wersji naszej aplikacji muzycznej MuzMachina wyposażonej w mechanizm „pogawędek” sieciowych Twój zespół może współpracować — możesz wysłać stworzoną kompozycję wraz z wiadomością tekstoną, a wszyscy biorący udział w pogawędce ją otrzymają. A zatem Twoje możliwości nie ograniczają się do *odczytywania* wiadomości przesyłanych przez inne osoby, w trywialny sposób — klikając wiadomość tekstoną — możesz wczytywać i odtwarzać nadsyłane kompozycje.

W tym rozdziale dowiesz się, czego wymaga stworzenie takiego klienta pogawędek.

Zdobędziemy także nieco informacji na temat tworzenia serwera *pogawędek*. Pełny kod aplikacji obsługującej muzyczne pogawędkи przedstawimy w części „Kod od kuchni”, jednak wcześniej napiszesz dwa inne programy, którym nadaliśmy nazwy: Śmiesznie prosty klient pogawędek oraz Bardzo prosty serwer pogawędek.

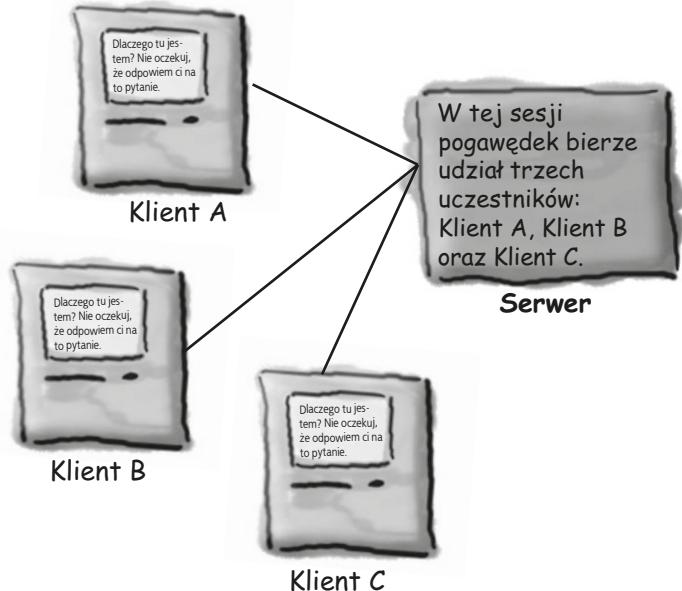
Możesz prowadzić najprawdziwszą, stymulującą intelektualnie pogawędkę. Wiadomości są wysypane do wszystkich osób biorących w niej udział.



Ogólne omówienie programu klienta

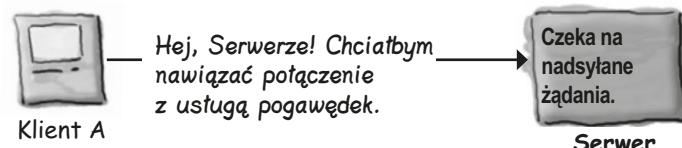
Klient musi wiedzieć o istnieniu serwera.

Serwer musi widzieć o istnieniu WSZYSTKICH klientów.

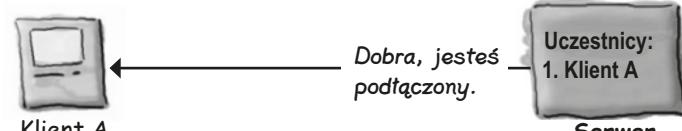


Jak to wszystko działa?

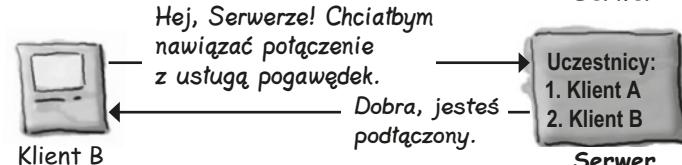
- 1 Klient nawiązuje połączenie z serwerem.



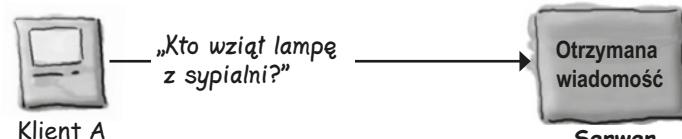
- 2 Serwer tworzy połączenie i dodaje klienta do listy uczestników pogawędki.



- 3 Kolejny klient nawiązuje połączenie.



- 4 Klient A wysyła wiadomość do serwera pogawędek.



- 5 Serwer rozsyła otrzymaną wiadomość do WSZYSTKICH uczestników pogawędki (w tym także do klienta, który ją wysłał).



Nawiązywanie połączenia, wysyłanie i odbieranie danych

Oto trzy zagadnienia, z którymi musimy się zapoznać, aby uruchomić naszego klienta pogawędek:

1) Nawiązywanie początkowego połączenia pomiędzy klientem a serwerem.

2) Wysyłanie wiadomości na serwer.

3) Odbieranie wiadomości nadsiłanych przez serwer.

Aby powyższe operacje mogły być przeprowadzane, należy wykonać bardzo wiele czynności niskiego poziomu. Mamy jednak szczęście, gdyż pakiet sieciowy (java.net) wchodzący w skład Java API sprawia, że z punktu widzenia programisty operacje sieciowe są wyjątkowo proste. W przedstawionych przykładach znajdziesz dużo kodu odpowiedzialnego za stworzenie i obsługę graficznego interfejsu użytkownika oraz znacznie mniej kodu realizującego operacje sieciowe i wejścia-wyjścia.

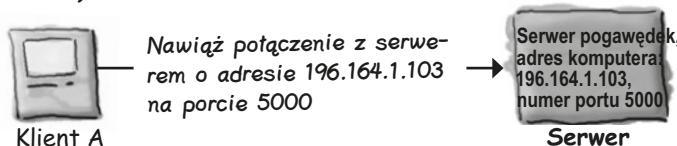
A to jeszcze nie wszystko.

Gdzieś w prostym przykładzie naszego klienta pogawędek czai się problem, z którym do tej pory jeszcze się nie spotkaliśmy — problem równoczesnego wykonywania dwóch czynności. Nawiązanie połączenia jest czynnością wykonywaną jednokrotnie (która może się zakończyć powodzeniem lub porażką).

Jednak potem klient pogawędek chce wysyłać wiadomości na serwer i jednocześnie odbierać wiadomości od innych uczestników pogawędk (rozsypane przez serwer). Hmm... to będzie wymagało poważniejszego przemyślenia, ale już niedługo do tego dojdziemy.

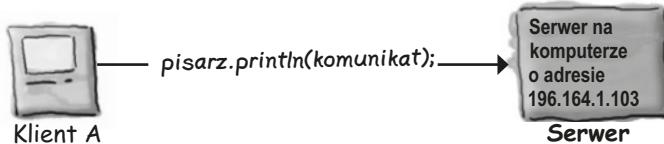
1 Połączenie

Klient łączy się z serwerem poprzez ustanowienie połączenia sieciowego (tak zwanego **gniazda**).



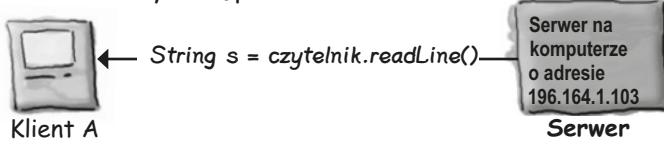
2 Wysłanie wiadomości

Klient wysyła wiadomość na serwer.



3 Odebranie wiadomości

Klient odbiera wiadomość wysłaną przez serwer.

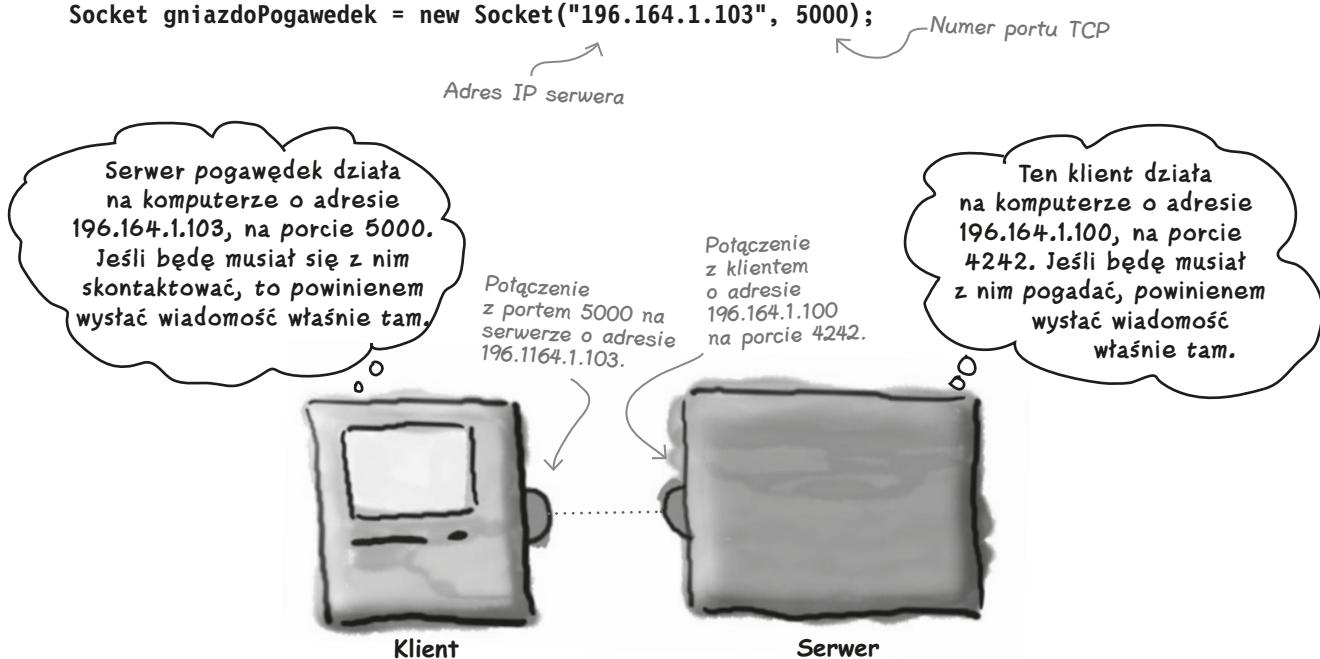


Nawiązanie połączenia sieciowego

Aby nawiązać połączenie z innym komputerem, będziemy potrzebowali tak zwanego gniazda. Gniazdo to obiekt klasy `java.net.Socket` reprezentujący połączenie sieciowe pomiędzy dwoma komputerami. A czym jest połączenie? To *relacja* pomiędzy dwoma komputerami, która sprawia, że **dwa programy wiedzą o swoim istnieniu**. Znacznie ważniejsze jest jednak to, iż programy te wiedzą, jak się wzajemnie *komunikować*. Innymi słowy, wiedzą, jak przesyłać pomiędzy sobą *bity*.

Nie będziemy się zajmować szczegółami niskiego poziomu, gdyż na szczeble są one obsługiwane przez mechanizmy znajdujące się na niższych poziomach „stosu sieciowego”. Nie przejmuj się, jeśli nie wiesz, czym jest „stos sieciowy”. To jedynie sposób wyobrażenia sobie warstw, przez które trzeba przekazać dane (bity), aby mogły zostać przesłane z programu napisanego w Javie i działającego w JVM na pewnym systemie operacyjnym, do fizycznych urządzeń sieciowych (takich jak karta sieciowa i kabel), a następnie dalej — na inny komputer. Ktoś musi się zająć tymi wszystkimi niewidocznymi szczegółami. Tym „kim” jest połączenie oprogramowania charakterystycznego dla systemu operacyjnego oraz sieciowego interfejsu programistycznego dostępnego w Javie. Tobie pozostaje obsługa operacji wysokiego — nawet *bardzo* wysokiego — poziomu, która jest zadziwiająco prosta. Nieprawdaż?

```
Socket gniazdoPogawedek = new Socket("196.164.1.103", 5000);
```



Istnienie połączenia sieciowego oznacza, że dwa komputery dysponują informacjami o sobie, w tym położeniem tego drugiego komputera (adresem IP) oraz numerem portu TCP.

Aby stworzyć połączenie sieciowe, musisz dysponować dwiema informacjami na temat serwera: **kim on jest** oraz na **jakim porcie** działa.

Innymi słowy, będziesz potrzebował adresu IP oraz numeru portu TCP.

Port TCP to tylko numer. 16-bitowa liczba identyfikująca konkretny program na serwerze

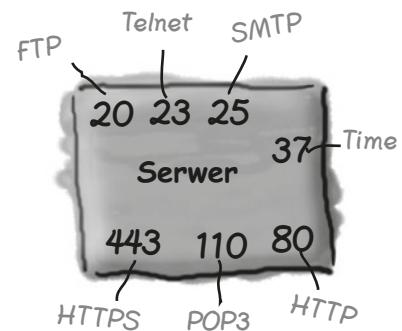
Twój serwer WWW (HTTP) działa na porcie numer 80. To standardowy port. Jeśli dysponujesz serwerem Telnet, to działa on na porcie 23. A FTP? Na porcie 20. A serwer poczty elektronicznej obsługujący protokół POP3? Na porcie 110. A serwer SMTP? Na porcie 25. Serwer czasu działa na porcie 37. Wyobraź sobie numery portów jako unikalne identyfikatory. Reprezentują one logiczne połaczenie z konkretnym oprogramowaniem działającym na serwerze. I to wszystko. Nawet jeśli rozkręcisz swój komputer i przejrzyś go dokładnie, to i tak nie znajdziesz portu TCP. A jednak serwer dysponuje aż 65 536 takimi portami (o numerach od 0 do 65 535). A zatem oczywiste jest, że nie reprezentują one miejsc, w których można by podłączyć fizyczne urządzenia. Są to jedynie liczby reprezentujące aplikacje.

Gdyby nie było numerów portów, serwer w żaden sposób nie mógłby określić, z jaką aplikacją klient chce nawiązać połaczenie. A ponieważ każda aplikacja może korzystać z unikalnego protokołu, wyobraź sobie, jakie byśmy mieli problemy. Na przykład, co by się stało, gdyby przeglądarka WWW połączyla się z serwerem poczty POP3, a nie z serwerem HTTP? Serwer pocztowy nie wie, jak należy analizować żądania HTTP! A nawet gdyby to potrafił, to i tak nie wiedziałby, jak należy obsługiwać żądania HTTP.

Pisząc program serwera, umieścisz w nim kod, który poinformuje program, na jakim porcie ma działać (w dalszej części rozdziału dowiesz się, jak należy to robić w Javie). Na potrzeby programu do prowadzenia pogawędek, który piszemy w tym rozdziale, wybraliśmy port 5000. Zrobiliśmy tak tylko dlatego, że ta liczba nam się spodobała. Jak również dla tego, iż liczba ta spełnia kryterium, w myśl którego numery portów muszą mieścić się w granicach do 1024 do 65 535. Dlaczego akurat 1024? Gdyż porty o numerach do 1023 zostały zarezerwowane dla powszechnie znanych usług, które wymieniliśmy wcześniej.

A jeśli tworzysz usługi (programy pełniące funkcję serwerów), które mają działać w sieci korporacyjnej, powinieneś skontaktować się z administratorem systemu i dowiedzieć się, jakie porty są już zajęte. Administrator może Cię poinformować, że, na przykład, nie możesz używać żadnego portu o numerze mniejszym niż 3000. Jeśli cenisz sobie swoje członki, to nie będziesz wykorzystywać takich numerów portów. Chyba że to jest Twoja sieć *domowa*. W takim wypadku musisz zapytać dzieci.

Powszechnie znane numery portów TCP używane przez popularne aplikacje serwerowe



Na komputerze może działać do 65 536 różnych aplikacji pełniących rolę serwerów, z których każda będzie używać jednego portu.

Porty TCP o numerach od 0 do 1023 są zarezerwowane dla powszechnie używanych usług. Nie używaj ich, tworząc własne programy serwerów*!

Serwer pogawędek, który napiszemy w tym rozdziale, używa portu 5000. Wybraliśmy liczbę z zakresu od 1024 do 65 535.

* Cóż, właściwie to możesz wykorzystać jeden z tych portów, ale administrator systemu komputerowego zapewne Cię zabije.

Nie istnieja
głupie pytania

P: Skąd znasz numer portu aplikacji serwera, z którą chcesz nawiązać połączenie?

O: To zależy, czy ta aplikacja jest jedną z powszechnie stosowanych usług. Jeśli próbujesz nawiązać połączenie z taką usługą (których przykłady podaliśmy na poprzedniej stronie — HTTP, SMTP, FTP itd.) możesz znaleźć numery portów w internecie (otwórz przeglądarkę Google i wpisz hasło „porty TCP”). Ewentualnie zapytaj zaprzyjaźnionego administratora z sąsiedztwa.

Jeśli jednak program nie jest jedną z takich usług, to będziesz musiał zapytać jego twórcę lub osobę, która go rozpowszechnia. Zazwyczaj, jeśli ktoś tworzy usługę sieciową i chce, aby inni tworzyli programy klienckie, które z niej korzystają, to publikuje wszystkie informacje na jej temat — adres IP, numer portu oraz używany protokół. Na przykład, jeśli chcesz napisać program klienta obsługujący serwer do gry w GO, to powinieneś zajrzeć na stronę WWW poświęconą temu serwerowi i na niej poszukać informacji o sposobie tworzenia klienta dla tego konkretnego serwera.

P: Czy więcej niż jeden program może działać na jednym porcie? Innymi słowy, czy dwie aplikacje działające na tym samym serwerze mogą mieć ten sam numer portu?

O: Niel. Jeśli spróbujesz skojarzyć program z portem, który już jest używany, to zostanie zgłoszony wyjątek `BindException`.

Skojarzenie programu z portem oznacza po prostu uruchomienie aplikacji i nakazanie jej używania konkretnego portu.Więcej na ten temat dowiesz się w dalszej części rozdziału poświęconej serwerom.

Numer portu to konkretny sklep w centrum handlowym

Adres IP to centrum handlowe.



Adres IP to jakby określenie konkretnego centrum handlowego, na przykład „Galeria Centrum”.



Numer portu to jakby określenie konkretnego sklepu, na przykład, „Sklep muzyczny Romka”.

WYŁĘŻ UMYŚŁ



W porządku, udało Ci się nawiązać połączenie. Klient i serwer znają swoje adresy IP i numery portów. Ale co teraz? W jaki sposób można komunikować się, wykorzystując to połączenie? Wyobraź sobie, jakie rodzaje komunikatów nasz klient pogawędka musi wysyłać i odbierać.



Właściwie jak te dwa programy mają się ze sobą komunikować?



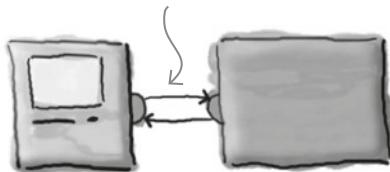
Program serwera pogawędka
Serwer

Odczyt danych z gniazda

Aby odczytywać dane z gniazda, należy użyć strumienia BufferedReader

Aby móc przekazywać dane poprzez połączenie, będziesz potrzebował strumieni. Starych, dobrych strumieni wejścia-wyjścia, takich jak te, których używaliśmy w poprzednim rozdziale. Jedną z najlepszych cech Javy jest to, iż większość operacji wejścia-wyjścia nie zwraca uwagi, do czego tak naprawdę są podłączone strumienie znajdujące się na początku łańcucha. Innymi słowy, możesz korzystać ze strumienia BufferedReader tak samo, jak byś odczytywał dane z pliku, choć tym razem używany strumień „połączeniowy” będzie operować nie na *pliku*, lecz na *gnieździe*.

Strumień wejściowy i wyjściowy, odpowiednio odczytujący i zapisujący dane w połączeniu sieciowym.



1 Nawiąż połączenie sieciowe z serwerem, tworząc w tym celu gniazdo.

```
Socket gniazdo = new Socket("127.0.0.1", 5000);
```

Numer portu, który znasz, gdyż POWIEDZIELISMY Ci, że nasz serwer pogawędek będzie działać na porcie 5000.

127.0.0.1 to adres IP reprezentujący tak zwany „localhost”, czyli ten sam komputer, na którym działa program. Możesz używać tego adresu podczas testowania klienta i serwera na tym samym komputerze.

2 Utwórz obiekt InputStreamReader połączony z wejściowym („połączeniowym”) strumieniem gniazda.

```
InputStreamReader strumien = new InputStreamReader(gniazdo.getInputStream());
```

InputStreamReader to „pomost” łączący bajtowy strumień niskiego poziomu (na przykład strumień odczytujący dane z gniazda) oraz znakowy strumień wysokiego poziomu (taki jak BufferedReader, który znajduje się na samym początku naszego łańcucha strumieni).

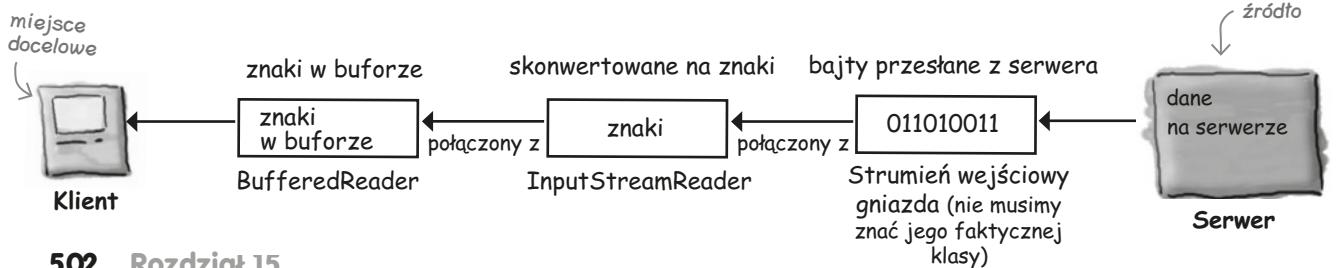
Jedyna, co musimy zrobić, to POPROSIĆ gniazda o zwrócenie strumienia wejściowego! Jest to strumień „połączeniowy” niskiego poziomu, jednak to nam nie przeszkadza, gdyż i tak połączymy go z innym strumieniem, który utatwia operacje na tekście.

Połącz strumień BufferedReader ze strumieniem InputStreamReader (który jest połączony ze strumieniem niskiego poziomu zwróconym przez gniazdo).

3 Utwórz strumień BufferedReader i odczytaj dane!

```
BufferedReader czytelnik = new BufferedReader(strumien);
```

```
String wiadomosc = czytelnik.readLine();
```



Aby zapisać dane w gnieździe, użyj strumienia PrintWriter

W poprzednim rozdziale nie używaliśmy strumienia PrintWriter, zamiast niego używaliśmy strumienia BufferedWriter. Możemy wybrać każdy z tych strumieni, jednak w przypadku zapisywania pojedynczych łańcuchów znaków standardowo stosowany jest strumień PrintWriter. Poza tym na pewno rozpoznasz dwie kluczowe metody tego strumienia — print() oraz println()! Zupełnie jak w starym, dobrym, standardowym strumieniu wyjściowym System.out.

1 Utwórz gniazdo reprezentujące połaczenie z serwerem.

```
Socket gniazdo = new Socket("127.0.0.1", 5000);
```

Ten fragment kodu jest taki sam jak na poprzedniej stronie — aby zapisywać dane na serwerze, musimy nawiązać z nim połaczenie.

2 Utwórz strumień PrintWriter połączony ze strumieniem wyjściowym gniazda (strumieniem „połączeniowym” niskiego poziomu).

```
PrintWriter pisarz = new PrintWriter(gniazdo.getOutputStream());
```

Strumień PrintWriter działa jak „pomost” pomiędzy danymi znakowymi oraz bajtami, które należy zapisywać w wyjściowym strumieniu gniazda. Dzięki połączeniu strumienia PrintWriter ze strumieniem wyjściowym gniazda, możemy zapisywać w gnieździe łańcuchy znaków.

Gniazdo zwraca strumień „połączeniowy” niskiego poziomu, który łączymy ze strumieniem PrintWriter (przekazując go w wywołaniu konstruktora strumienia PrintWriter).

3 Zapisz coś.

```
pisarz.println("Wiadomość do wysłania"); ← Metoda println() dodaje znak nowego wiersza na końcu wysyłanego łańcucha znaków.  
pisarz.print("Kolejna wiadomość"); ← Metoda print() nie dodaje znaku nowego wiersza.
```



Program CodziennePoradyKlient

Zanim zaczniemy tworzyć program pogawędek sieciowych, zaczniemy do czegoś prostszego. „Doradca” to aplikacja serwera udzielająca praktycznych i inspirujących porad, dzięki którym łatwiej będzie Ci przetrwać długie dni kodowania.

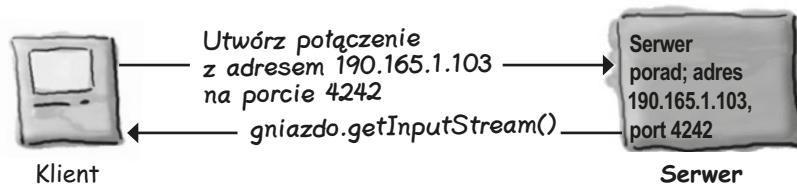
Stworzymy klienta dla aplikacji Doradca, który pobiera z serwera porady za każdym razem, gdy nawiążę z nim połączenie.

Na co czekasz? Kto *wie*, jakie okazje możesz stracić, jeśli nie będziesz dysponować tą aplikacją.



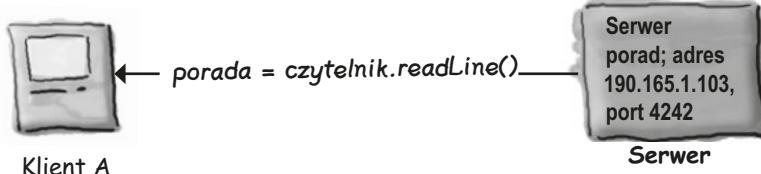
1 Nawiązanie połączenia.

Klient nawiązuje połączenie z serwerem i pobiera strumień wejściowy.



2 Odczyt.

Klient odczytuje wiadomość z serwera.



Kod programu CodziennePoradyKlient

Przedstawiony poniżej program tworzy gniazdo (obiekt Socket), następnie tworzy strumień BufferedReader (wykorzystując przy tym także i inne strumienie), po czym odczytuje pojedynczy wiersz znaków wysłany przez aplikację serwera (jakikolwiek program działający na porcie 4242).

```

import java.io.*;
import java.net.*;

public class CodziennePoradyKlient {

    public void doDziela() {
        try {
            Socket s = new Socket("127.0.0.1", 4242);
            InputStreamReader strCzytelnik = new InputStreamReader(s.getInputStream());
            BufferedReader czytelnik = new BufferedReader(strCzytelnik);

            String porada = czytelnik.readLine();
            System.out.println("Dziś powinieneś: " + porada);

            czytelnik.close();
        } catch(IOException ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        CodziennePoradyKlient klient = new CodziennePoradyKlient();
        klient.doDziela();
    }
}

```

W tym fragmencie kodu może wystąpić wiele nieprzewidzianych problemów.

Nawiązujemy połączenie z programem działającym na porcie 4242, na tym samym komputerze, na którym jest uruchomiony ten program (czyli na komputerze „localhost”).

Łączymy strumień BufferedReader ze strumieniem InputStreamReader oraz ze strumieniem wejściowym gniazda.

To wywołanie zamyka WSZYSTKIE strumienie.

Ta metoda działa DOKŁADNIE tak samo, jak gdybyś używał strumienia BufferedReader do odczytu danych z PLIKU. Innymi słowy, w chwili, gdy będziesz się postygować metodami tej klasy, strumień nie będzie ani wiedzieć, ani zwracać uwagi na to, skąd pochodzą znaki.

Połączenia sieciowe



Zaostrz ołówek

Sprawdź, czy dobrze zapamiętałeś klasy i strumienie wykorzystywane przy odczytzie i zapisie danych w gniazdach. Postaraj się nie zaglądać na poprzednie strony!

Aby **odczytać** tekst z gniazda:



Klient A



Napisz lub narysuj tańcuch strumieni używanych przez klienta do odczytania danych przesyłanych z serwera

Serwer

Aby **wysłać** dane do gniazda:



Klient A



Napisz lub narysuj tańcuch strumieni używanych przez klienta, aby wysłać dane na serwer

Serwer



Zaostrz ołówek

Wypełnij puste pola:

Jakich dwóch informacji potrzebuje klient, aby nawiązać połączenie sieciowe z serwerem?

Jakie numery portów TCP są zarezerwowane dla powszechnie znanych usług, takich jak HTTP lub FTP?

Prawda czy fałsz? Zakres poprawnych numerów portów TCP można wyrazić przy użyciu liczby całkowitej typu short.

Tworzenie prostego serwera

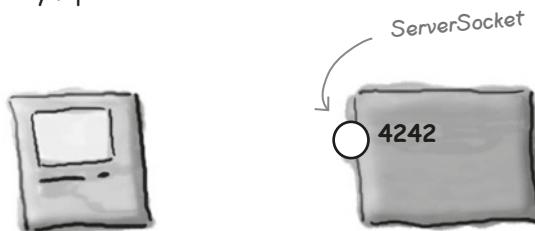
A zatem co trzeba zrobić, aby napisać aplikację serwera? Całe zadanie sprowadza się do stworzenia pary gniazd. Tak, pary — czyli dwóch. Pierwsze z nich — `ServerSocket` — oczekuje na żądania klientów (zgłaszone przez klienta z chwilą wywołania konstruktora `new ServerSocket()`), a drugie — zwyczajne gniazdo (obiekt `Socket`), służy do komunikacji z klientem.

Jak to działa?

- 1 Aplikacja serwera tworzy gniazdo `ServerSocket` na określonym porcie:

```
ServerSocket gniazdoSrw = new ServerSocket(4242);
```

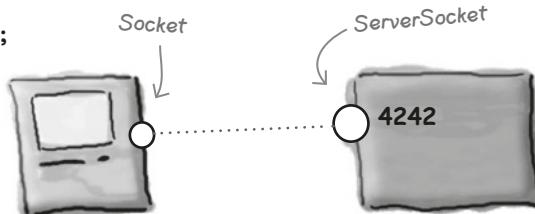
W ten sposób aplikacja zaczyna oczekiwac
na żądania kierowane na port o numerze 4242.



- 2 Klient tworzy połączenie z aplikacją serwera:

```
Socket gniazdo = new Socket("190.164.1.103", 4242);
```

Klient zna adres IP oraz numer portu (opublikowany
lub przekazany przez osobę, która skonfigurowała
aplikację serwera tak, aby ta działała na określonym porcie).

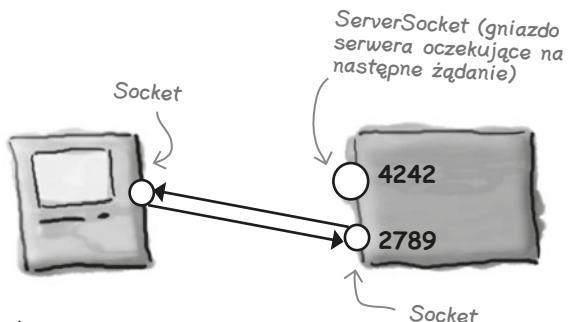


- 3 Serwer tworzy nowe gniazdo służące do komunikacji z klientem.

```
Socket gniazdo = gniazdoSrw.accept();
```

Metoda `accept()` blokuje działanie programu i wznowia go dopiero po odebraniu żądania połączenia przesłanego przez klienta. Kiedy klient spróbuje nawiązać połączenie, metoda ta zwróci zwyczajne gniazdo (operujące na innym porcie), które będzie wiedzieć, jak można prowadzić komunikację z klientem (czyli zna adres IP oraz numer portu klienta).

Zwrócony obiekt `Socket` operuje na innym porcie niż gniazdo serwera — `ServerSocket` — dzięki czemu gniazdo serwera ponownie może oczekiwac na żądania zgłoszane przez inne klienty.



Kod aplikacji CodziennePoradySerwer

Przedstawiony poniżej program tworzy gniazdo ServerSocket i oczekuje na nadysiane żądania. Kiedy takie żądanie zostanie odebrane (czyli gdy klient wywoła konstruktor new Socket(), próbując nawiązać połączenie z tą aplikacją), nasz serwer utworzy nowe gniazdo służące do komunikacji z klientem. Następnie serwer tworzy strumień PrintWriter (wykorzystując przy tym strumień wyjściowy gniazda) i przesyła wiadomość do klienta.

```

import java.io.*;      Pamiętaj o zimportowaniu
import java.net.*;    niezbędnych pakietów.

public class CodziennePoradySerwer {          Porady są wybierane z tej tablicy.

    String[] listaPorad = {"Używaj mniejszych bitów", "Chodź w dopasowanych spodniach. Nie, te nie sprawiają, że wyglądasz grubo.", "Jedno słowo: nieodpowiednie", "Tylko dziś - bądź uczciwy - powiedz swojemu szefowi, co *naprawdę* czujesz", "Może chcesz zastanowić się nad swoją fryzurą."};

    public void doDziela() {
        try {
            ServerSocket gniazdoSrw = new ServerSocket(4242);          Gniazdo ServerSocket sprawia, że
                                                                       nasza aplikacja serwera „oczekuje” na
                                                                       żądania przesypane na port o numerze
                                                                       4242, na komputerze, na którym
                                                                       aplikacja została uruchomiona.

            while(true) {          Aplikacja serwera zaczyna realizować nieskończoną
                                  pętlę, w której oczekuje (i obsługuje) żądania
                                  nadysywane przez klientów.

                Socket gniazdo = gniazdoSrw.accept(); <-- Metoda accept() wstrzymuje działanie
                                                               programu aż do momentu odebrania
                                                               żądania, a wtedy zwraca zwyczajne
                                                               gniazdo (operujące na jakimś
                                                               nieznanym porcie) umożliwiające
                                                               prowadzenie komunikacji z klientem.

                PrintWriter pisarz = new PrintWriter(gniazdo.getOutputStream());
                String porada = wybierzPorade();
                pisarz.println(porada);
                pisarz.close();
                System.out.println(porada);
            }
        } catch(IOException ex) {
            ex.printStackTrace();
        }
    } // koniec metody

    public static void main(String[] args) {
        CodziennePoradySerwer serwer = new CodziennePoradySerwer();
        serwer.doDziela();
    }

    private String wybierzPorade() {
        int random = (int) (Math.random() * listaPorad.length);
        return listaPorad[random];
    }
}

```

(Pamiętaj, że tańcuchy znaków umieszczone w tej tablicy zostały wyświetcone w kilku wierszach, gdyż nie zmieściły się w jednym. Nigdy nie naciskaj klawisza Enter w środku tańcucha znaków!)

Teraz używamy gniazda reprezentującego połączenie z klientem do utworzenia strumienia PrintWriter i przestania do klienta (przy użyciu metody println()) tańcucha znaków zawierającego poradę. Następnie zamkamy gniazdo, gdyż obsługa żądania została zakończona.



WYTĘŻ UMYSŁ

Skąd serwer wie, w jaki sposób ma się komunikować z klientem?

Klient zna adres IP oraz numer portu serwera, jednak jak to się dzieje, że serwer jest w stanie nawiązać połączenie i komunikować się z klientem (oraz utworzyć strumień wyjściowy i wejściowy)?

Zastanów się, jak, skąd oraz kiedy serwer zdobywa niezbędne informacje na temat klienta.

Nie istniejąca grupa pytania

P: **Kod serwera przedstawiony na poprzedniej stronie ma BARDZO poważne ograniczenie — otóż wydaje się, że może obsługiwać tylko jedno żądanie naraz!**

O: Tak to prawda. Nasz serwer nie może odebrać żądania od klienta, dopóki nie zakończy obsługi bieżącego żądania i nie rozpocznie kolejnej iteracji nieskończonej pętli while (w której program zatrzyma się na wywoaniu accept() i będzie oczekiwany na zgłoszenie żądania, a gdy takie żądanie odbierze, utworzy połączenie z nowym klientem i ponownie rozpocznie cały proces).

P: **Zadam pytanie nieco inaczej: Jak stworzyć aplikację serwera, która może współbieżnie obsługiwać wiele żądań??? Przedstawione rozwiązanie nigdy nie zdałoby egzaminu w przypadku tworzenia, na przykład, serwera pogawędek.**

O: O, to jest naprawdę proste. Należy użyć niezależnych wątków i do każdego z nich przekazać nowe gniazdo. Już za chwilę dowiesz się, jak to zrobić.



CELNE SPOSTRZEŻENIA

- Klient i serwer komunikują się przy użyciu połączenia sieciowego reprezentowanego przez gniazda.
- Gniazdo reprezentuje połączenie pomiędzy dwiema aplikacjami, które mogą (choć nie muszą) działać na dwóch różnych komputerach.
- Klient musi znać adres IP (lub nazwę domeny) oraz numer portu TCP, którego używa aplikacja serwera.
- Port TCP to 16-bitowa liczba całkowita bez znaku przypisywana aplikacji serwera. Numery portów TCP pozwalają różnym klientom na nawiązywanie połączenia z tym samym komputerem, a jednocześnie zapewniają, że klienci będą się komunikować z różnymi aplikacjami serwerów.
- Porty o numerach od 0 do 1023 są zarezerwowane dla „powszechnie znanych” usług, takich jak HTTP, FTP, SMTP i tak dalej.
- Klient nawiązuje połączenie z serwerem, tworząc gniazdo (obiekt klasy Socket):


```
Socket g = new Socket("127.0.0.1", 4200);
```
- Po nawiązaniu połączenia klient możliwe pobrać z gniazda strumień wyjściowy i wejściowy. Są to strumienie „połączeniowe” niskiego poziomu.

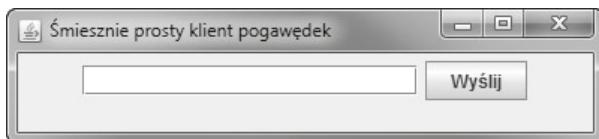

```
g.getInputStream();
g.getOutputStream();
```
- Aby odczytać dane tekstowe przesypane z serwera, należy stworzyć strumień BufferedReader połączony ze strumieniem InputStreamReader, który z kolei jest połączony ze strumieniem wejściowym gniazda.
- InputStreamReader jest „pomostem”, który pobiera bajty i zamienia je na dane tekstowe (czyli na znaki). Zazwyczaj jest on stosowany jako środkowe ogniwo łańcucha strumieni łączące strumień wysokiego poziomu (taki jak BufferedReader) ze strumieniem niskiego poziomu (na przykład ze strumieniem wejściowym gniazda).
- Aby przesłać dane na serwer, należy stworzyć strumień PrintWriter połączony bezpośrednio ze strumieniem wyjściowym gniazda. Następnie można wysyłać łańcuchy znaków na serwer, wywołując metody println() lub print().
- Aplikacja serwera używa gniazd ServerSocket do oczekiwania i odbierania żądań nadawanych przez klienty na port o określonym numerze.
- Kiedy gniazdo ServerSocket odbierze żądanie, „akceptuje” je, zwracając obiekt Socket reprezentujący połączenie z klientem.

Tworzenie klienta pogawędek

Aplikację klienta pogawędek napiszemy w dwóch etapach. W pierwszym z nich stworzymy wersję, która wysyła wiadomości na serwer, lecz nie ma zamiaru odczytywać wiadomości nadsyłanych przez innych uczestników pogawędki (to eksytuujący i nieoczekiwany zwrot w całej koncepcji pogawędek sieciowych).

W drugim etapie rozwiniemy nasz program i stworzymy klienta pogawędek, który zarówno wysyła wiadomości, jak i je odbiera.

Wersja pierwsza. Tylko wysyłanie wiadomości



Wpisz wiadomość i kliknij przycisk „Wyślij”, aby ją wysłać na serwer. W tej wersji programu nie będziemy odbierać żadnych wiadomości z serwera, dlatego jego interfejs graficzny nie zawiera wielowierszowego pola tekstowego.

Ogólna struktura kodu:

```
public class ProstyKlientPogawedekA {  
  
    JTextField wiadomosc;  
    PrintWriter pisarz;  
    Socket gniazdo;  
  
    public void dodziela() {  
        // tworzymy graficzny interfejs użytkownika i rejestrujemy odbiorcę  
        // zdarzeń w przycisku, wywołujemy metodę konfigurującą mechanizmy  
        // sieciowe  
    } // koniec metody  
  
    private void konfigurujKomunikacje() {  
        // tworzymy gniazdo (Socket), a następnie strumień PrintWriter  
        // zapisujemy strumień w składowej o nazwie pisarz.  
    } // koniec metody  
  
    public class PrzyciskWyslijListener implements ActionListener {  
        public void actionPerformed(ActionEvent ev) {  
            // pobieramy tekst z pola tekstowego i wysyłamy na serwer, używając  
            // w tym celu pisarza (PrintWriter)  
        }  
    } // koniec klasy wewnętrznej  
} // koniec klasy zewnętrznej
```

```

import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

Instrukcje importu zapewniające
dostęp do klasy strumieni (java.io),
gniazd (java.net) oraz graficznego
interfejsu użytkownika.

public class ProstyKlientPogawedekA {

    JTextField wiadomosc;
    PrintWriter pisarz;
    Socket gniazdo;

    public void doDziela() {
        JFrame ramka = new JFrame("Śmiesznie prosty klient pogawędek");
        JPanel panelGlowny = new JPanel();
        wiadomosc = new JTextField(20);
        JButton przyciskWyslij = new JButton("Wyślij");
        przyciskWyslij.addActionListener(new PrzyciskWyslijListener());
        panelGlowny.add(wiadomosc);
        panelGlowny.add(przyciskWyslij);
        konfigurujKomunikacje();
        ramka.getContentPane().add(BorderLayout.CENTER, panelGlowny);
        ramka.setSize(400,90);
        ramka.setVisible(true);
    } // koniec metody

    private void konfigurujKomunikacje() {
        try {
            gniazdo = new Socket("127.0.0.1", 5000);
            pisarz = new PrintWriter(gniazdo.getOutputStream());
            System.out.println("obsługa sieci gotowa do użycia");
        } catch(IOException ex) {
            ex.printStackTrace();
        }
    } // koniec metody

    public class PrzyciskWyslijListener implements ActionListener {
        public void actionPerformed(ActionEvent ev) {
            try {
                pisarz.println(wiadomosc.getText());
                pisarz.flush();
            } catch(Exception ex) {
                ex.printStackTrace();
            }
            wiadomosc.setText("");
            wiadomosc.requestFocus();
        }
    } // koniec klasy wewnętrznej

    public static void main(String[] args) {
        new ProstyKlientPogawedekA().doDziela();
    }
} // koniec klasy zewnętrznej

```

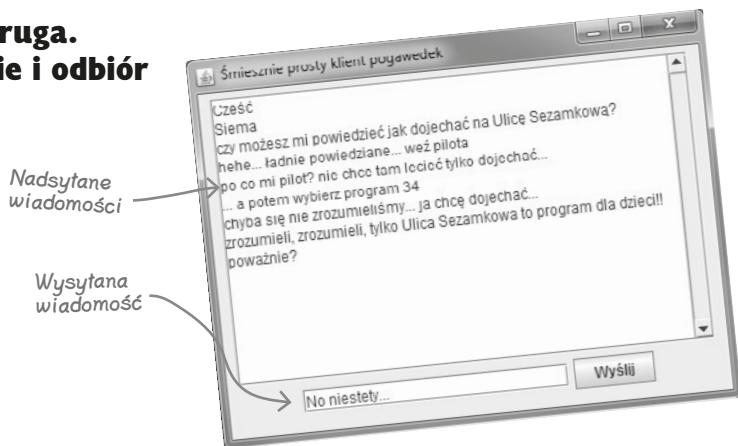
Używamy adresu IP reprezentującego „localhost”, aby można było przetestować klienta i serwer na tym samym komputerze.

To właśnie w tym fragmencie kodu tworzymy gniazdo (Socket) oraz strumień PrintWriter. (Kod ten jest wywoływany z poziomu metody doDziela(), tuż przed wyświetleniem interfejsu graficznego aplikacji).

Jeśli już teraz chcesz wypróbować, jak działa ten program, wpisz kod podany pod koniec tego rozdziału, w części zatytułowanej „Kod gotowy do użycia”. Najpierw w jednym oknie wiersza poleceń uruchom serwer, a następnie w drugim oknie uruchom klienta.

Usprawnianie klienta pogawędek

Wersja druga. Wysyłanie i odbiór



Jak tylko serwer odbierze wiadomość, zaraz rozsyła ją do wszystkich klientów biorących udział w pogawędce. Kiedy klient wysyła wiadomość, zostanie ona wyświetlona w obszarze prezentującym otrzymane wiadomości dopiero wtedy, gdy serwer rozsła ją do wszystkich klientów.

Ważne pytanie: W JAKI sposób odbierzesz wiadomość przesyłaną z serwera?

To powinno być proste — konfigurując operacje sieciowe, wystarczy dodatkowo utworzyć strumień wejściowy (prawdopodobnie będzie to strumień BufferedReader). Pozwoli on na odczytywanie wiadomości przy wykorzystaniu metody readLine().

Jeszcze ważniejsze pytanie: KIEDY należy odbierać wiadomości przesyłane z serwera?

Zastanów się nad tym. Jakie mamy możliwości?

1 Rozwiążanie pierwsze. Sprawdzanie, czy na serwerze są nowe wiadomości co 20 sekund.

Zalety: Cóż, to można zrobić.

Wady: Skąd serwer ma wiedzieć, które wiadomości już widziałeś, a których jeszcze nie zdążyłeś odebrać? W takim przypadku serwer nie mógłby stosować strategii „roześlij-i-zapomnij” za każdym razem, gdy odbierze wiadomość, lecz musiałby przechowywać wszystkie odebrane wiadomości. Poza tym dlaczego co 20 sekund? Takie opóźnienie może poważnie ograniczyć użyteczność naszej aplikacji, z kolei zmniejszenie opóźnienia spowoduje zwiększenie niepotrzebnych odwołań do serwera. To rozwiązanie jest nieefektywne.

2 Rozwiążanie drugie. Odczytywanie czegoś z serwera za każdym razem, gdy użytkownik wyśle wiadomość.

Zalety: Wykonalne, bardzo proste.

Wady: To rozwiązanie jest po prostu głupie. Niby dlaczego sprawdzać wiadomości nadsybane przez innych uczestników akurat w tak określonych momentach? A co, jeśli użytkownik lubi się ukrywać i nie wysyła żadnych własnych wiadomości?

3 Rozwiążanie trzecie. Odczytywanie wiadomości jak tylko zostaną rozesłane przez serwer.

Zalety: Najbardziej efektywne i najbardziej użyteczne.

Wady: W jaki sposób można jednocześnie wykonać dwie czynności? Gdzie można umieścić taki kod? Gdzieś trzeba by umieścić pętlę, która zawsze czekałaby na wiadomości rozsypane przez serwer i była gotowa do ich odczytania. Ale gdzie taką pętlę umieścić? Po uruchomieniu graficznego interfejsu użytkownika nic się nie dzieje aż do momentu zgłoszenia jakiegoś zdarzenia przez jeden z komponentów.



Już pewnie wiesz, że wybierzymy trzecie rozwiązanie.

Chcemy, aby pewien fragment kodu działał nieprzerwanie i sprawdzał, czy na serwerze pojawiły się jakieś nowe wiadomości, *a jednocześnie, aby operacje te nie przeszkadzały w korzystaniu z graficznego interfejsu aplikacji!* A zatem chcemy, aby „coś”, w sposób niewidoczny dla użytkownika, odczytywało nowe wiadomości rozsypane przez serwer, podczas gdy użytkownik będzie mógł wpisywać własną wiadomość lub przeglądać listę odebranych wiadomości.

A to oznacza, że w końcu będziemy potrzebowali nowego wątku. A zatem także nowego i niezależnego stosu.

Chcemy, aby wszystkie operacje wykonywane w pierwszej wersji programu działały dokładnie tak samo, a jednocześnie, aby istniał jakiś dodatkowy *proces*, który w tym samym czasie odbiera wiadomości rozsypane przez serwer i wyświetla je w wielowierszowym polu tekstowym.

Cóż, to nie dzieje się dokładnie tak. Jeśli nie dysponujesz komputerem wyposażonym w kilka procesorów, to każdy nowy wątek uruchamiany przez Javę nie będzie się tak naprawdę stawał niezależnym procesem wykonywanym przez system operacyjny. Jednak będzie *sprawiać wrażenie*, jak gdyby był takim niezależnym procesem.

Używając Javy, naprawdę możesz jednocześnie spacerować i żuć gumę

Wielowątkowość w Javie

W Javie wielowątkowość jest jednym z integralnych mechanizmów języka, a stworzenie nowego wątku jest bardzo szybkie i proste:

```
Thread w = new Thread();  
w.start();
```

I to wszystko. Tworząc nowy obiekt Thread, uruchomisz nowy wątek realizacji programu posiadający swój własny stos.

Ale jest jeden problem.

Taki wątek niczego nie robi, a zatem „umiera” niemal w tej samej chwili, w której został uruchomiony. Kiedy wątek „umiera”, jego nowy stos znika bezpowrotnie. I tak się kończy ta opowieść.

A zatem umknęło nam podstawowe zagadnienie — *praca*, jaką wątek ma wykonać. Innymi słowy, jest nam potrzebny kod, który ma być wykonywany w nowym, niezależnym wątku.

Wielowątkowość z jakiej możemy korzystać w Javie sprawia, że musimy się przyjrzeć zarówno samym *wątkom*, jak i *zadaniom*, które te wątki *realizują*. Musimy się także przyjrzeć *klasie Thread* należącej do pakietu *java.lang*. (Pamiętasz, że *java.lang* to pakiet, który jest zawsze importowany — w niejawnym sposób — i w którym zostały umieszczone wszystkie klasy mające kluczowe znaczenie dla języka, takie jak *String* lub *System*).

Możliwość stosowania wielu wątków w Javie zapewnia jedna klasa — Thread

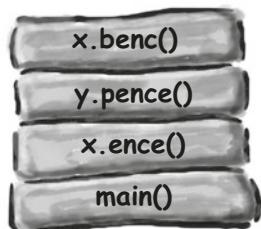
Mówiąc o wielowątkowości, słowa „wątek” możemy używać w dwóch znaczeniach. W pierwszym „wątek” to „niezależny wątek realizacji” programu, czyli niezależny stos wywołań. Jednak „wątek” może także oznaczać obiekt klasy Thread, należącej do pakietu `java.lang`. Obiekt Thread reprezentuje niezależny wątek realizacji; takie obiekty będziesz tworzyć za każdym razem, gdy będziesz chciał uruchomić nowy wątek, czyli niezależnie wykonywany fragment kodu.

Wątek to niezależny „wątek realizacji”. Innymi słowy — niezależny stos wywołań.

Thread to klasa Javy, która reprezentuje wątki.

Aby utworzyć wątek, należy stworzyć obiekt klasy Thread.

wątek



wątek główny



innego wątka uruchomionego przez program

Wątek

Thread
<code>void join()</code>
<code>void start()</code>
<code>static void sleep()</code>

klasa `java.lang.Thread`

Wątek, to niezależny wątek realizacji. A to oznacza osobny stos wywołań. Każda aplikacja Javy uruchamia „wątek główny” — czyli wątek, w którym na samym spodzie stosu wywołań znajduje się metoda `main()`. Za uruchomienie wątku głównego odpowiada wirtualna maszyna Javy (ona także odpowiada za uruchamianie innych wątków, jeśli uzna to za konieczne, przykładem takiego wątku może być wątek odśmiecacz). Ty — jako programista — możesz uruchamiać nowe wątki programowo.

Thread to klasa reprezentująca wątki realizacji. Udostępnia metody do uruchamiania wątku, łączenia kilku wątków oraz „usypiania” wątku. (Nie są to co prawda wszystkie metody tej klasy, jednak jedne z najważniejszy, a co ważniejsze, właśnie tych metod będziemy już niedługo potrzebować).

Jakie są konsekwencje posiadania więcej niż jednego stosu wywołań?

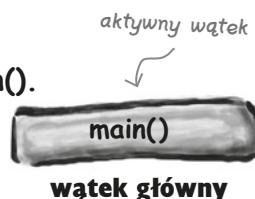
Dysponując więcej niż jednym stosem wywołań, uzyskujemy *wrażenie* jednoczesnego wykonywania kilku różnych czynności. W rzeczywistości naprawdę równoczesne wykonywanie różnych operacji jest możliwe jedynie na komputerze wieloprocesorowym, niemniej jednak dzięki wątkom Javy można uzyskać *wrażenie* współbieżnego wykonywania kilku czynności. Innymi słowy proces wykonywania programu może być tak szybko przełączony z jednego stosu wywołań na drugi, iż użytkownik będzie mieć wrażenie, jak gdyby oba te stosoły były wykonywane jednocześnie. Pamiętaj, że Java jest jedynie procesem realizowanym przez właściwy system operacyjny komputera, a zatem przede wszystkim sama Java musi być „procesem aktualnie wykonywanym” przez system operacyjny. Kiedy jednak nadziejcie czas na realizację Javy, to właściwie *co* będzie wykonywać jej wirtualna maszyna? Jakie kody bajtowe? Te, które znajdują się na samym wierzchołku aktualnie wykonywanego stosołu! A w ciągu 100 milisekund aktualnie wykonywany kod może się zmienić w zupełnie inną metodę umieszczoną na *innym* stosołku wywołań.

Jedną z rzeczy, na jaką wątek musi zwracać uwagę, jest to, która instrukcja jest aktualnie wykonywana na stosołku tego wątku (oraz do jakiej metody ta instrukcja należy).

Proces uruchamiania nowego wątku może wyglądać mniej więcej tak:

1 Wirtualna maszyna Javy wywołuje metodę main().

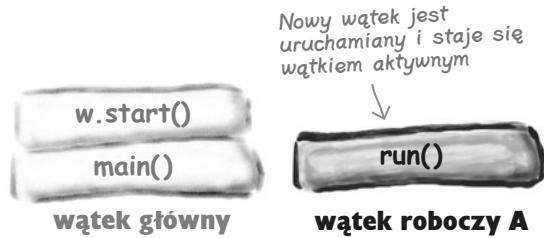
```
public static void main(String[] args) {  
    ...  
}
```



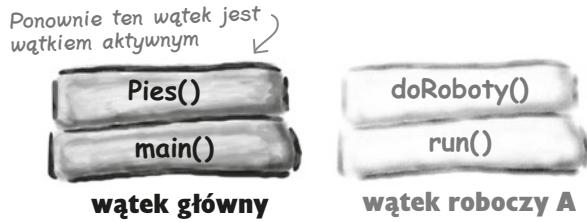
2 Metoda main() uruchamia nowy wątek. Realizacja wątku głównego jest chwilowo wstrzymywana, a rozpoczyna się wykonywanie nowego wątku.

```
Runnable r = new ZadanieMojegoWątku();  
Thread w = new Thread(r);  
w.start();  
Pies p = new Pies();
```

Już za chwilę dowiesz się, co to wszystko oznacza...



3 Wirtualna maszyn Javy zmienia aktualnie wykonywany wątek — na przemian jest nim nowy wątek (wątek roboczy A) i wątek główny. To zmienianie wątków trwa aż do momentu, gdy oba zostaną zakończone.



Oto jak można stworzyć i uruchomić nowy wątek:

1 Stwórz obiekt Runnable (zadanie realizowane w wątku)

```
Runnable zadanie = new MojeZadanie();
```

Runnable to interfejs, który zostanie przedstawiony na następnej stronie. Napiszesz klasę implementującą ten interfejs, w której należy zdefiniować zadania, jakie wątek będzie miał wykonać. Innymi słowy, stworzysz metodę, która zostanie uruchomiona na nowym stosie wywołań.



2 Utwórz obiekt Thread (pracownik) i przekaż do niego obiekt klasy implementującej interfejs Runnable (zadanie do wykonania)

```
Thread mojWatek = new Thread(zadanie);
```

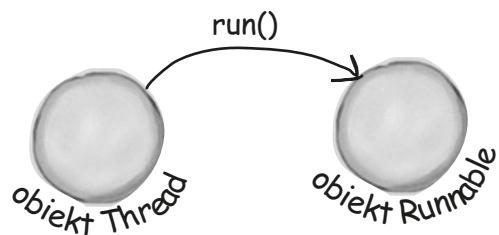
W wywołaniu konstruktora klasy Thread przekaż nowy obiekt klasy implementującej interfejs Runnable. W ten sposób przekazujemy wątkowi informację o tym, jaką metodę należy umieścić na samym spodzie nowego stosu — będzie to metoda run() obiektu Runnable.



3 Uruchom wątek

```
mojWatek.start();
```

Do momentu wywołania metody start() obiektu Thread nic się nie wydarzy. To właśnie wywołanie tej metody sprawia, że zwyczajny obiekt Thread „zamienia się” w niezależny wątek realizacji. W chwili uruchomienia nowego wątku na samym spodzie jego stosu umieszczana jest metoda run() wskazanego obiektu Runnable.



Każdy wątek potrzebuje zadania, które będzie wykonywać, metody, którą będzie można umieścić na stosie nowego wątku



Runnable jest dla wątku tym, czym praca dla pracownika. Obiekt Runnable określa zadanie, jakie wątek ma wykonać.

Obiekt Runnable ma metodę, która jest umieszczana na samym spodzie stosu nowego wątku — jest to metoda `run()`.

Obiekt Thread potrzebuje pracy. Zadania, które wykona wątek, kiedy zostanie uruchomiony. To zadanie jest pierwszą metodą, która zostanie umieszczona na stosie nowego wątku, co więcej, metoda ta zawsze musi wyglądać następująco:

```
public void run() {
    // kod, który będzie wykonywany w nowym wątku
}
```

Skąd wątek, wie jaką metodę należy umieścić na spodzie stosu? Metoda ta jest znana, gdyż określa ją kontrakt interfejsu Runnable. Zadanie, jakie wątek ma wykonać, może być zdefiniowane w dowolnej klasie implementującej interfejs Runnable. Wątek zwraca uwagę tylko na to, czy w wywołaniu konstruktora klasy Thread przekażesz obiekt klasy reprezentującej interfejs Runnable.

Przekazując taki obiekt w wywołaniu konstruktora klasy Thread, dajesz obiekowi Thread możliwość wywołania metody `run()`. Innymi słowy, określasz zadanie, jakie wątek ma wykonać.

Interfejs Runnable definiuje tylko jedną metodę — `public void run()`.
(Pamiętaj, Runnable to interfejs,
zatem metoda będzie publiczna
niezależnie od sposobu, w jaki ja
zapiszesz).

Aby stworzyć zadanie dla wątku, zaimplementuj interfejs Runnable

Interfejs Runnable należy do pakietu java.lang, zatem nie trzeba go importować.

```
public class MojeZadanie implements Runnable {
```

```
    public void run() {
        doDziela();
    }
```

Interfejs Runnable zawiera tylko jedną metodę, którą należy zaimplementować, jest to metoda `public void run()` (jak widać, metoda nie pobiera żadnych argumentów). To właśnie w tej metodzie należy podać ZADANIE, które wątek ma wykonać. Metoda `run()` zostanie umieszczona na samym spodzie nowego stosu.

```
    public void doDziela() {
        kolejnaRobota();
    }
```

```
    public void kolejnaRobota() {
        System.out.println("Wierzchołek stosu!");
    }
}
```

```
class WatkiTester {
```

```
    public static void main(String[] args) {
        Runnable zadanieWatku = new MojeZadanie();
        Thread mojWatek = new Thread(zadanieWatku);
    }
}
```

Nowy obiekt `Runnable` przekaż w wywołaniu konstruktora klasy `Thread`. W ten sposób określisz, jaką metodę wątek ma umieścić na spodzie nowego stosu. Innymi słowy, określasz w ten sposób pierwszą metodę, która wywoła nowy wątek.

1

```
mojWatek.start(); ←
System.out.println("Z powrotem w metodzie main()");
```

Nowy wątek realizacji nie zostanie utworzony aż do momentu wywołania metody `start()` obiektu `Thread`. Można by rzec, że dopiero po uruchomieniu wątek staje się „prawdziwym” wątkiem. Zanim zostanie uruchomiony, jest jedynie obiektem `Thread` podobnym do wszelkich innych obiektów — obiektem, który nie wykazuje żadnych cech wątku.

}

1

```
mojWatek.start()
main()
```

wątek główny

2

```
kolejnaRobota()
doDziela()
run()
```

nowy wątek

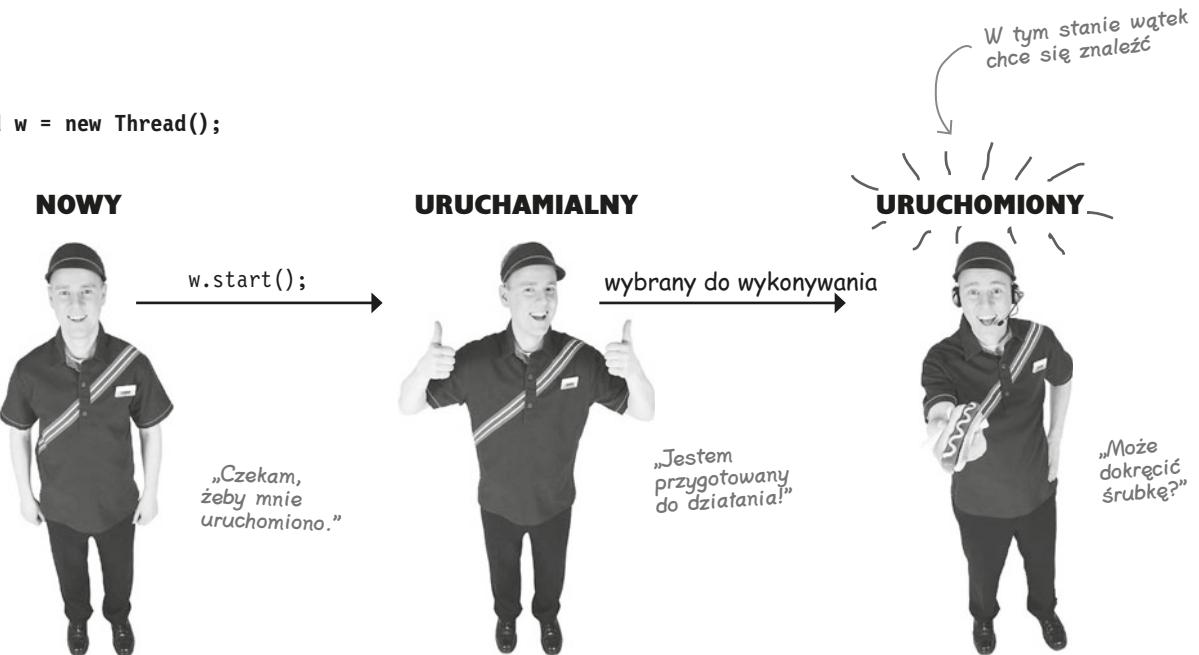


WYTĘŻ UMYSŁ

Jak myślisz, jakie będą wyniki wykonania klasy `WatkiTester`? (Przedstawimy je w dalszej części rozdziału).

Trzy stany nowego wątku

Thread w = new Thread();



Thread w = new Thread(r);

Obiekt Thread został utworzony, lecz nie uruchomiony. Innymi słowy, dysponujemy *obiektem Thread*, lecz nie *wątkiem realizacji*.

`w.start();`

Po uruchomieniu wątku przechodzi on do stanu określonego jako uruchomialny. Oznacza to, że wątek jest gotowy do działania i tylko czeka na swoją „wielką chwilę” — czyli aż zostanie wybrany i rozpocznie pracę. W tym momencie istnieje już nowy stos, który będzie używany przez ten wątek.

To jest stan, o którym śniały wszystkie wątki! Aby stać się Wybrańcem. Wątkiem aktualnie wykonywanym. Tę decyzję może podjąć jedynie mechanizm zarządzający wątkami. Możesz czasami *wpływać* na tę decyzję, jednak nie możesz wymusić zmiany stanu wątku z uruchomialny na uruchomiony. Jeśli wątek znajduje się w stanie uruchomiony, to ten wątek (i TYLKO on) posiada aktywny stos wywołań, a metoda znajdująca się na wierzchołku tego stosu jest wykonywana.

Ale to nie wszystko. Kiedy wątek zostanie już uznany za uruchomialny, to będzie mógł zmieniać stan na: uruchomiony, uruchomialny lub na stan dodatkowy, który można by określić jako chwilowo nie uruchomialny (lub po prostu „zablokowany”).

Stany wątków

Typowa pętla stanów — uruchamialny-uruchomiony

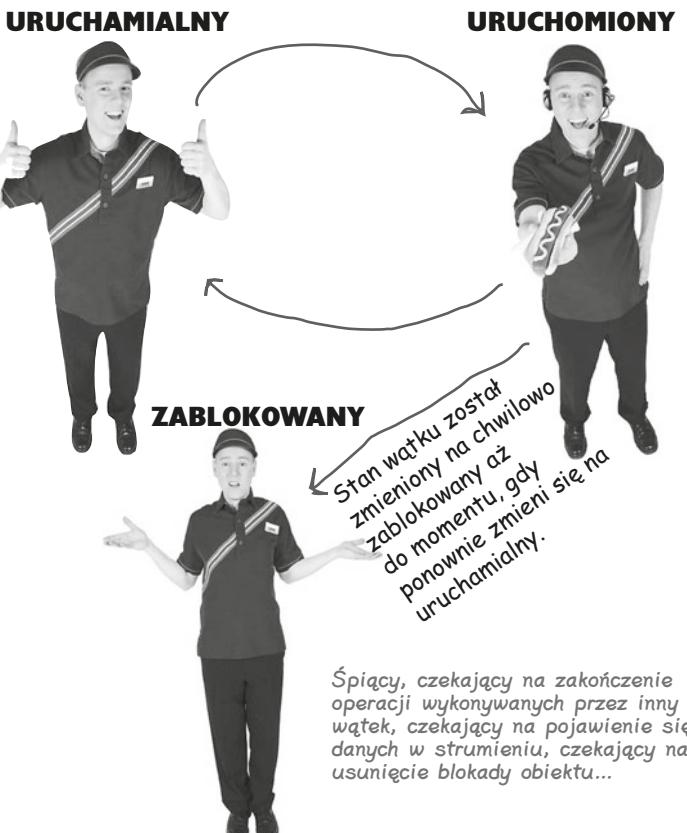
Zazwyczaj stan wątku zmienia się z uruchamialny na uruchomiony i na odwrót. Pierwsza z tych zmian stanu następuje, gdy mechanizm zarządzający wątkami wybierze wątek, który ma być aktualnie wykonywany; natomiast druga — gdy mechanizm zarządzający wyrzuci aktualnie wykonywany wątek na koniec kolejki i da szansę działania innemu wątkowi.



Wątek można chwilowo zablokować

Istnieje wiele powodów, dla których mechanizm zarządzający wątkami może zmienić stan wątku na zablokowany. Na przykład wątek może wykonywać kod odczytujący dane ze strumienia wejściowego gniazda, lecz w strumieniu nie ma żadnych danych, które można by odczytać. W takim przypadku mechanizm zarządzający zmieni stan wątku z uruchomiony na jakiś inny, aż do momentu, gdy pojawią się jakieś dane. Może się też zdarzyć, że to wykonywany kod zażąda „uśpienia” wątku (poprzez wywołanie metody `sleep()`), albo wątek musi poczekać, gdyż chciał wywołać metodę „zablokowanego” obiektu. W takim przypadku wątek musi czekać, aż wątek, który miał blokadę obiektu, usunie ją.

Wszystkie z tych warunków (choć może ich być znacznie więcej) powodują chwilowe zablokowanie wątku.



Mechanizm zarządzający wątkami

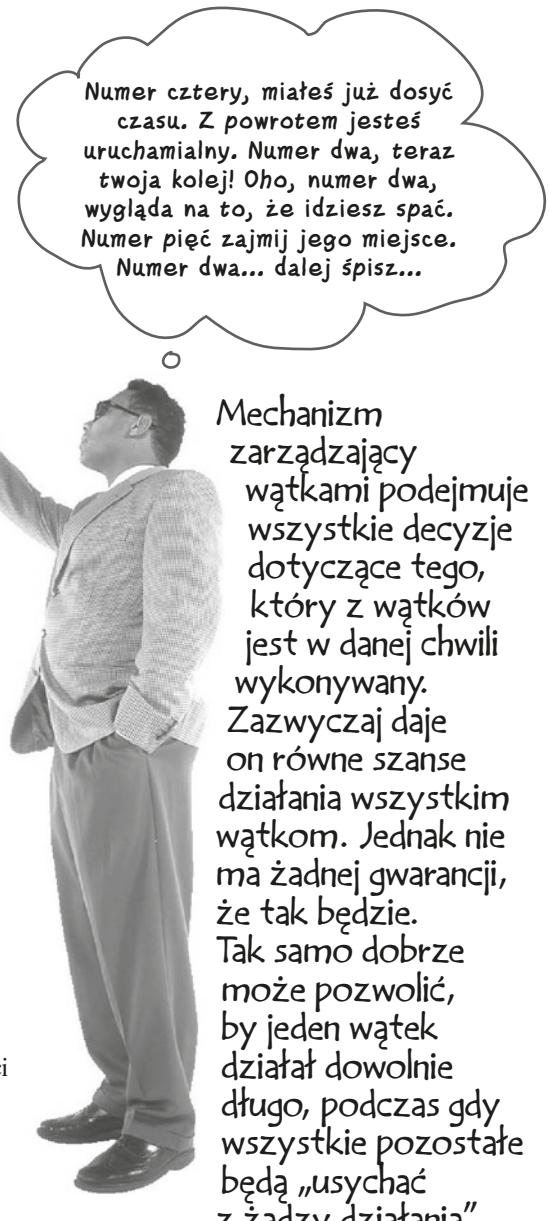
Mechanizm zarządzający podejmuje wszystkie decyzje odnośnie tego, jaki wątek przejdzie ze stanu wykonywalny do wykonywany oraz kiedy (i w jakich okolicznościach) jego stan się ponownie zmieni. Mechanizm ten określa, jaki wątek należy uruchomić, na jak długo oraz co się z nim potem stanie.

Nie możesz kontrolować poczynań mechanizmu zarządzającego. Nie ma żadnego interfejsu programistycznego, który umożliwiałby wywoływanie jego metod. A co najważniejsze, nie ma żadnych gwarancji odnośnie decyzji podejmowanych przez ten mechanizm! (Choć w niektórych przypadkach sposoby działania tego mechanizmu można określić jako „niemal gwarantowane”, to jednak trzeba położyć duży nacisk na słowo „niemal”).

W efekcie oznacza to, że *nie należy uzależniać poprawności działania programu od konkretnego zachowania mechanizmu zarządzającego wątkami!* W wirtualnych maszynach Javy mechanizm ten jest implementowany na różne sposoby, a nawet uruchomienie tego samego programu na tym samym komputerze może dać różne rezultaty. Jednym z najgorszych błędów popełnianych przez początkujących programistów Javy jest testowanie programów wielowątkowych na jednym komputerze i założenie, że mechanizm zarządzający wątkami zawsze będzie działać w ten sam sposób, niezależnie od tego, gdzie program będzie wykonywany.

A zatem co to oznacza dla idei „napisz raz, uruchamiaj wszędzie”? Otóż oznacza to, że aby stworzyć kod niezależny od platformy, na jakiej będzie uruchamiany, Twój wielowątkowy program musi działać *niezależnie* od zachowania mechanizmu zarządzającego wątkami. A to oznacza, iż program nie może zakładać, że, na przykład, wszystkie wątki będą wykonywane sprawiedliwie — czyli że będą uruchamiane tak samo często i będą działać tak samo długo. Choć w dzisiejszych czasach jest to raczej bardzo mało prawdopodobne, to jednak może się zdarzyć, że Twój program zostanie uruchomiony na wirtualnej maszynie Javy wyposażonej w mechanizm zarządzający, który ogłoszi: „W porządku, wątek numer pięć, twoja kolej, a jeśli o mnie chodzi, to możesz działać aż do zakończenia twojej metody run()”.

Niemal we wszystkich sytuacjach kluczem do powodzenia jest *usypianie*. Tak, właśnie *usypianie*. Uśpienie wątku nawet na kilka milisekund powoduje zmianę stanu aktualnie wykonywanego wątku i danie możliwości działania innemu wątkowi. Metoda sleep() daje nam tylko jedną gwarancję — że usypiany wątek *nie* przejdzie do stanu wykonywany przed upłynięciem czasu określonego w wywoaniu metody. Na przykład, jeśli każesz uśpić wątek na 2 sekundy (czyli 2000 milisekund), to wątek ten nigdy nie przejdzie do stanu uruchomiony *przed* upłynięciem co najmniej tych dwóch sekund.

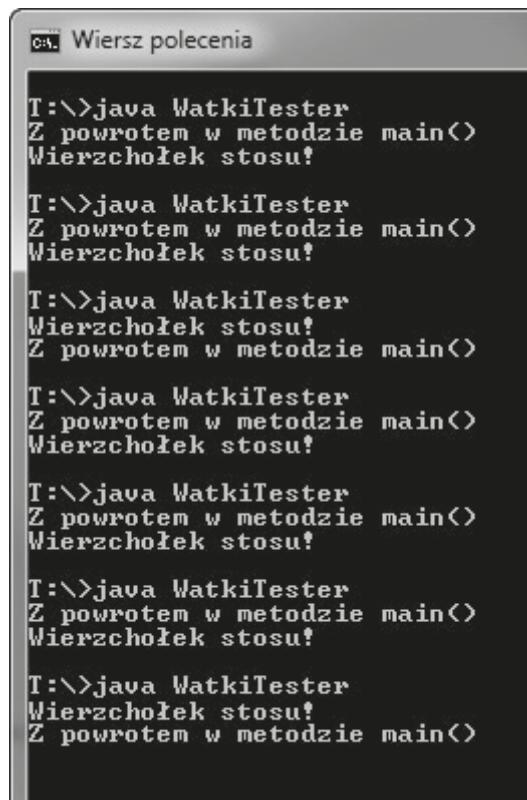


Przykład pokazujący jak nieprzewidywalne mogą być decyzje podejmowane przez mechanizm zarządzający wątkami...

Uruchomienie poniższego programu na tym samym komputerze:

```
class MojeZadanie implements Runnable {  
    public void run() {  
        doDziela();  
    }  
  
    public void doDziela() {  
        kolejnaRobota();  
    }  
  
    public void kolejnaRobota() {  
        System.out.println("Wierzchołek stosu!");  
    }  
}  
  
class WatkiTester {  
    public static void main(String[] args) {  
        Runnable zadanieWatku = new MojeZadanie();  
        Thread mojWatek = new Thread(zadanieWatku);  
  
        mojWatek.start();  
  
        System.out.println("Z powrotem w metodzie main()");  
    }  
}
```

Może dać następujące rezultaty:

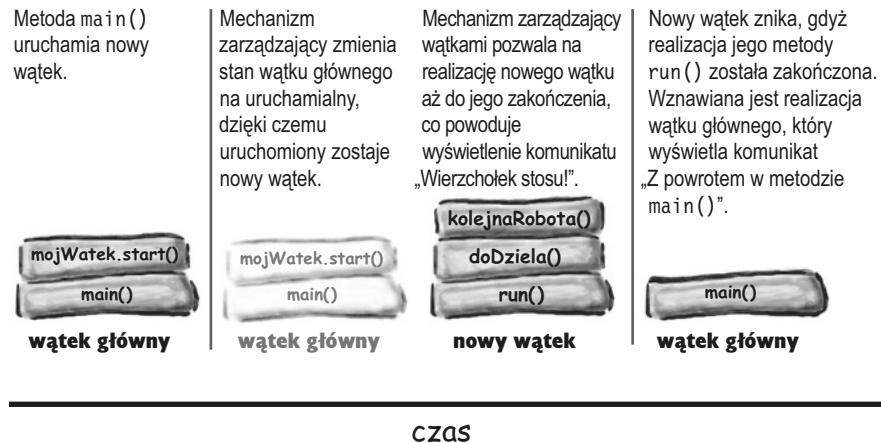


```
T:\>java WatkiTester  
Z powrotem w metodzie main()  
Wierzchołek stosu!  
  
T:\>java WatkiTester  
Z powrotem w metodzie main()  
Wierzchołek stosu!  
  
T:\>java WatkiTester  
Wierzchołek stosu!  
Z powrotem w metodzie main()  
  
T:\>java WatkiTester  
Z powrotem w metodzie main()  
Wierzchołek stosu!  
  
T:\>java WatkiTester  
Z powrotem w metodzie main()  
Wierzchołek stosu!  
  
T:\>java WatkiTester  
Z powrotem w metodzie main()  
Wierzchołek stosu!  
  
T:\>java WatkiTester  
Wierzchołek stosu!  
Z powrotem w metodzie main()
```

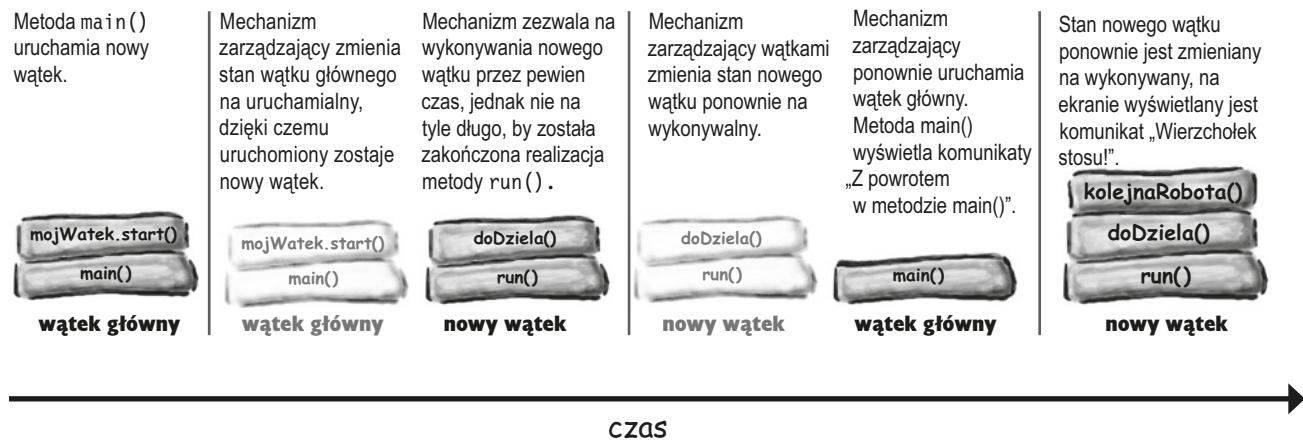
Zauważ, że kolejność komunikatów zmienia się w sposób losowy. Czasami to nowy wątek zostanie zakończony jako pierwszy, a czasami pierwszy będzie wątek główny.

Jak to się stało, że uzyskaliśmy różne wyniki?

Czasami program jest wykonywany w następujący sposób:



Natomiast czasami realizacja programu przebiega w następujący sposób:



| Nie istniejąca
grupie pytania

P: Widziałem przykłady, w których nie tworzone odrębnej klasy implementującej interfejs Runnable, lecz tworzono klasę potomną klasy Thread i przesłaniano w niej metodę run(). W ten sposób, podczas tworzenia nowego obiektu Thread, można użyć konstruktora bezargumentowego:

```
Thread w = new Thread(); // interfejs Runnable nie
                         // jest używany
```

O: Tak, to jest inny sposób tworzenia nowego wątku. Jednak przeanalizuj go z obiektowego punktu widzenia. W jakim celu tworzone są klasy potomne? Pamiętaj, że w tym przypadku rozmawiamy o dwóch różnych sprawach — *wątku* (obiekcie Thread) oraz *zadaniu*, jakie ten wątek ma wykonać. Z obiektowego punktu widzenia są to dwie zupełnie niezależne sprawy i należy je umieścić w odrębnych klasach. Jedyną sytuacją, w której będziesz chciał tworzyć klasę potomną klasy Thread, jest tworzenie nowego i bardziej wyspecjalizowanego typu wątku. Jeśli wyobrażisz sobie wątek jako pracownika, to pomyśl, że nie powinieneś rozszerzać klasy Thread, chyba że potrzebujesz pracownika o bardziej wyspecjalizowanych umiejętnościach. Jeśli jednak chodzi Ci tylko o to, aby wątek (pracownik) wykonał nowe zadanie, to powinieneś zaimplementować interfejs Runnable w nowej klasie; innymi słowy, powinieneś dostosować klasę do *zadania*, a nie do *pracownika*.

Takie rozwiązanie jest jedynie kwestią projektu, nie ma natomiast nic wspólnego z efektywnością działania programu ani z możliwościami samego języka. Stworzenie klasy potomnej klasy Thread i przesłanie w niej metody run() jest całkowicie poprawne i dozwolone. Niemniej jednak rzadko kiedy rozwiązanie to będzie używane.

P: Czy można wielokrotnie używać tego samego obiektu Thread? Czy można przydzielić mu nowe zadanie, a następnie ponownie uruchomić, wywołując metodę start()?

O: Nie. Po wykonaniu metody run() wątek nie będzie można ponownie uruchomić. W rzeczywistości po zakończeniu metody run() wątek przechodzi do stanu, o którym jeszcze nie wspominaliśmy — staje się *martwy*. W takim martwym wątku realizacja metody run() została zakończona i już nigdy nie będzie można ponownie go uruchomić. Obiekt Thread znajdujący się w takim stanie wciąż istnieje na stercie, można więc wywoływać jego pozostałe metody (o ile to będzie właściwe), jednak taki obiekt bezpowrotnie straci wszystkie cechy wątku. Innymi słowy, nie ma już niezależnego stosu wywołań, a obiekt Thread nie jest już wątkiem. Jest jedynie obiektem podobnym do wszystkich innych obiektów.

Istnieją jednak wzorce projektowe określające sposoby tworzenia puli wątków, których można używać do wykonywania różnego typu operacji. Niemniej jednak nie robi się tego poprzez ponowne uruchamianie martwego wątku.



CELNE SPOSTRZEŻENIA

- Wątek to „niezależny wątek realizacji”.
- Każdy wątek w Javie dysponuje własnym stosem wywołań.
- Klasa Thread — `java.lang.Thread` — to klasa, której obiekty reprezentują wątki.
- Obiekty Thread potrzebują zadania, które mogłyby wykonać. Takim zadaniem jest obiekt pewnej klasy, która implementuje interfejs Runnable.
- Interfejs Runnable ma tylko jedną metodę — `run()`. To właśnie ta metoda zostaje umieszczone na samym spodzie nowego stosu wywołań. Innymi słowy, jest to pierwsza metoda wykonywana w nowym wątku.
- Aby utworzyć nowy wątek, należy przekazać obiekt Runnable w wywołaniu konstruktora klasy Thread.
- Po utworzeniu obiektu Thread, lecz przed wywołaniem jego metody `start()`, wątek znajduje się w stanie **NOWY**.
- W momencie rozpoczęcia wątku (czyli wywołania metody `start()` obiektu Thread) tworzony jest nowy stos, a na jego spodzie zostaje umieszczone metoda `run()`. W tym momencie wątek znajduje się w stanie **URUCHAMIALNY** i oczekuje na uruchomienie.
- Wątek staje się **URUCHOMIONY**, gdy mechanizm zarządzający wątkami „wybierze” go i uzna za aktualnie wykonywany. Na komputerze wyposażonym w jeden procesor w każdej chwili może być wykonywany tylko jeden wątek.
- Czasami stan wątku może się zmienić z **URUCHOMIONY** na **ZABLOKOWANY** (czyli „czasowo niewykonywany”). Wątek może zostać zablokowany, ponieważ oczekuje na dane ze strumienia, został uśpiony lub czeka na dostęp do zablokowanego obiektu.
- Nie ma żadnych gwarancji, że mechanizm zarządzający wątkami będzie działać w jakikolwiek konkretny sposób. A zatem nie można mieć pewności, że wszystkie wątki będą „sprawiedliwie” wykonywane po kolei. Możesz jednak mieć wpływ na mechanizm zarządzający wątkami i ułatwić mu takie cykliczne zmienianie wykonywanych wątków; w tym celu powinieneś co pewien czas usypiać wątki.

Usypianie wątku

Jeśli chcesz wspomóc zmienianie wykonywanych wątków, to najlepszym sposobem, aby to zrobić, jest usypianie aktualnie wykonywanego wątku. Jedyną rzeczą, jaką należy w tym celu wykonać, jest wywołanie metody `sleep()` i przekazanie do niej czasu (wyrażonego w milisekundach), na jaki należy uśpić wątek.



Na przykład wywołanie:

`Thread.sleep(2000);`

sprawi, że wątek przestanie być wykonywany i nie wróci do stanu uruchomiony przedniej niż po upływie dwóch sekund. Wątek *nie może* ponownie przejść do stanu uruchomiony wcześniej niż po upływie co najmniej dwóch sekund.

Tak się nieszczęśliwie składa, że wywołanie metody `sleep()` może zgłosić wyjątek `InterruptedException`. Jest to wyjątek sprawdzany, co sprawia, że wywołania tej metody muszą być umieszczone wewnątrz bloku `try-catch` (albo że wyjątek należy zadeklarować). A zatem, wywołanie metody `sleep()` zazwyczaj wygląda następująco:

```
try {
    Thread.sleep(2000);
} catch (InterruptedException ex) {
    ex.printStackTrace();
}
```

Prawdopodobnie *nigdy* się nie zdarzy, że „odpoczynek” Twojego wątku zostanie przerwany; wyjątek ten został wprowadzony w interfejsie programistycznym Javy po to, aby wspomóc mechanizm komunikacji wątków, którego w praktyce niemal nikt nie używa. Nie zmienia to jednak faktu, iż musisz postępować zgodnie z zaleceniem „obsługuj lub deklaruj”, a zatem będziesz musiał przyzwyczaić się do umieszczania wywołań metody `sleep()` w blokach `try-catch`.

Teraz wiesz, że wątek nie zostanie uruchomiony *przed* upłynięciem podanego czasu, ale czy jest możliwe, że zostanie uruchomiony *później*? I tak, i nie.

Tak naprawdę, to nie ma to większego znaczenia, gdyż po „obudzeniu” **wątek zawsze przechodzi do stanu uruchamialny!** A zatem nigdy się nie zdarzy, że bezpośrednio po wyjściu ze stanu uśpienia wątek zacznie być wykonywany. Gdy wątek się „obudzi”, ponownie znajduje się na łasce mechanizmu zarządzającego. W przypadku aplikacji, które nie wymagają perfekcyjnej synchronizacji czasowej i dysponują tylko kilkoma wątkami, może się wydawać, że wątek wychodzi ze stanu uśpienia i zaczyna być wykonywany dokładnie wtedy, gdy zaplanowano (na przykład po 2000 milisekund). Jednak nie możesz zakładać, że tak się stanie i uzależniać od tego działania Twojego programu.

„Uśpij” wątek,
jeśli chcesz mieć
pewność, że inne
wątki uzyskają
możliwość
działania.

Po wyjściu ze stanu uśpienia wątek zostaje uznany za uruchamialny i oczekuje, aż mechanizm zarządzający ponownie go uruchomi.

Usypanie wątków w celu zapewnienia bardziej przewidywalnego działania programu

Czy pamiętasz ten przykładowy program, który za każdym razem generował inne wyniki? Spójrz jeszcze raz na stronę 524 i przeanalizuj zarówno kod programu, jak i jego wyniki. Czasami metoda `main()` musi czekać na zakończenie nowego wątku (i wyświetlenie komunikatu „Wierzchołek stosu!”, a czasami nowy wątek wraca do stanu uruchamialny, zanim się zakończy, co pozwala na ponowne uruchomienie wątku głównego, który wyświetla komunikat „Z powrotem w metodzie `main()`”. W jaki sposób można rozwiązać ten problem? Przerwij na chwilę lekturę i spróbuj odpowiedzieć na pytanie, gdzie należałoby umieścić wywołanie metody `sleep()`, aby zapewnić, że komunikat „Z powrotem w metodzie `main()`” zawsze będzie wyświetlany przed komunikatem „Wierzchołek stosu!”.

Poczekamy, aż znajdziesz odpowiedź na to pytanie (istnieje kilka rozwiązań, które przyniosą zamierzony efekt).

Już wiesz?

```
public class MojeZadanie implements Runnable {
    public void run() {
        doDziela();
    }

    public void doDziela() {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }

    kolejnaRobota(); ←
}

public void kolejnaRobota() {
    System.out.println("Wierzchołek stosu!");
}

class WatkiTester {
    public static void main(String[] args) {
        Runnable zadanieWatku = new MojeZadanie();
        Thread mojWatek = new Thread (zadanieWatku);
        mojWatek.start();
        System.out.println("Z powrotem w metodzie main()");
    }
}
```

Wywołanie metody `sleep()` w tym miejscu sprawi, że wykonywanie wątku zostanie wstrzymane.

Po wywołaniu metody `sleep()` aktualnie wykonywanym wątkiem stanie się ponownie wątek główny, który wyświetli komunikat „Z powrotem w metodzie `main()`”. Następnie, po okolo dwusekundowej przerwie, zostanie wykonany ten wiersz kodu, który wywoła metodę `kolejnaRobota()`, co doprowadzi do wyświetlenia komunikatu „Wierzchołek stosu!”.

Oto, do czego dajemy — zwróć uwagę na kolejność wyświetlonych komunikatów:

```
java WatkiTester
Z powrotem w metodzie main()
Wierzchołek stosu!

java WatkiTester
Z powrotem w metodzie main()
Wierzchołek stosu!

java WatkiTester
Z powrotem w metodzie main()
Wierzchołek stosu!

java WatkiTester
Z powrotem w metodzie main()
Wierzchołek stosu!

java WatkiTester
Z powrotem w metodzie main()
Wierzchołek stosu!
```

Tworzenie i uruchamianie dwóch wątków

Wątki mają nazwy. Każdemu wątkowi możesz nadać dowolnie wybraną nazwę, ewentualnie możesz przyjąć ich nazwy domyślne. Jedną z najlepszych cech nazw wątków jest to, iż można ich użyć w celu określenia, który wątek jest aktualnie wykonywany. W poniższym przykładzie tworzone są dwa wątki. Każdy z nich ma to samo zadanie — działać w pętli i w każdej iteracji wyświetlać nazwę aktualnie wykonywanego wątku.

```

public class Watki2 implements Runnable {
    public static void main(String[] args) {
        Watki2 program = new Watki2(); ← Tworzymy obiekt klasy
        Thread alfa = new Thread(program); ← implementującej Runnable.
        Thread beta = new Thread(program); ← Tworzymy dwa wątki mające tę samą implementację
        alfa.setName("Wątek Alfa"); ← interfejsu Runnable (czyli realizujące to samo
        beta.setName("Wątek Beta"); ← zadanie — w dalszej części rozdziału dokładniej
        alfa.start(); ← Określamy nazwy wątków.
        beta.start(); ← Rozpoczynamy realizację wątków.
    }

    public void run() {
        for (int i = 0; i < 25; i++) {
            String nazwaWątku = Thread.currentThread().getName();
            System.out.println("Aktualnie działa: " + nazwaWątku);
        }
    }
}

```

Każdy wątek wykona tę pętlę, wyświetlając swoją nazwę podczas każdej iteracji.

Fragment wyników generowanych przez program, w którym pętle wątków wykonują 25 iteracji.

Co się stanie?

Czy wątki będą się zmieniać kilkukrotnie? Czy nazwy wątków będą wyświetlane w różnej kolejności? Jak często wątki będą się zmieniać? Czy zmiana będzie następować po każdej iteracji? A może po pięciu iteracjach?

Już znasz odpowiedź na to pytanie — *nie wiadomo!* Wszystko zależy od mechanizmu zarządzającego wątkami. W zależności od używanego przez Ciebie systemu operacyjnego, wersji wirtualnej maszyny Javy i posiadanego procesora możesz uzyskać zupełnie inne wyniki.

W przypadku uruchomienia w systemie MacOS X 10.2 (Jaguar), przy pięciu lub mniejszej ilości iteracji, najpierw kończy się wykonywanie wątku Alfa, a potem wątku Beta. Taki przebieg realizacji programu był powtarzalny — bardzo powtarzalny. Oczywiście nie mieliśmy żadnych gwarancji, ale wyniki były powtarzalne.

Jeśli jednak zwiększymy ilość iteracji pętli do 25, to wyniki zaczynają się zmieniać. Wątek Alfa może nie zdążyć wykonać wszystkich 25 iteracji do momentu, gdy mechanizm zarządzający przerwie jego wykonywanie i uruchomi wątek Beta.

Wiersz polecenia	
Aktualnie działa:	Wątek Beta
Aktualnie działa:	Wątek Alfa
Aktualnie działa:	Wątek Beta
Aktualnie działa:	Wątek Alfa
Aktualnie działa:	Wątek Beta
Aktualnie działa:	Wątek Alfa

Czyż wątki nie są wspaniałe?



Cóż... Tak, możemy. JEST pewien problem związanego ze stosowaniem wątków

To „zagadnienia” związane ze współbieżnością.

Współbieżna realizacja może doprowadzić do współzawodnictwa.

Współzawodnictwo prowadzi natomiast do uszkodzenia danych. Uszkodzenie danych wzbudza strach... a resztę już znasz.

Wszystko to sprowadza się do jednego potencjalnie śmiertelnie niebezpiecznego scenariusza istnienia dwóch lub kilku wątków mających dostęp do *danych* jednego obiektu. Innymi słowy, chodzi o sytuację, w której dwie metody wykonywane na dwóch różnych stosach jednocześnie wywołują metody jednego obiektu na stercie, na przykład jego metody ustawiające i zwracające.

Cały ten problem można by porównać do sytuacji w której „prawa ręka nie wie, co robi lewa ręka”. Oto mamy dwa wątki, które nie zwracają uwagi na nic innego i beztrosko wykonują swoje metody. Każdy z nich uważa, że to właśnie on jest „tym jedynym”, i że tylko on się liczy. W końcu, kiedy wątek przestaje być wykonywany i przechodzi do stanu uruchamialny (lub zablokowany), to jest tak, jakby tracił przytomność. Kiedy ponownie zacznie być wykonywany, nie wie, że jego realizacja została przerwana.

Problem małżeński Czy tę parę da się uratować?

Następni w specjalnym programie Doktora Stefana

[Transkrypcja epizodu nr 42]

Witamy w programie Doktora Stefana.



Dzisiejsza opowieść będzie dotyczyć dwóch najczęściej podawanych powodów rozwodów. Chodzi oczywiście o finanse i sen.

Oto współczesna para, której postaramy się pomóc w rozwiązaniu ich problemów — Robert i Monika. Para ta ma wspólne łóżko i konto bankowe. Jednak jeśli nie uda nam się znaleźć rozwiązania, to ich stadło nie potrwa już długo. A jaki jest problem Moniki i Roberta? To już klasyka — „dwie osoby, jedno konto”.

Oto jak ich problem przedstawia Monika:

„Robert i ja zgodziłyśmy się, że żadne z nas nie będzie doprowadzać do przekroczenia limitu na naszym wspólnym koncie. Zatem stosujemy następującą procedurę: ktokolwiek wyciąga pieniądze z konta, przed wykonaniem operacji musi najpierw sprawdzić jego stan. Wszystko wydawało się takie proste. Aż tu nagle okazuje się, że nasze czekи nie mają pokrycia, a my musimy płacić odsetki za przekroczenie limitu!”

Myślałam, że to jest niemożliwe. Myślałam, że nasza procedura jest bezpieczna. Ale wtedy stało się coś takiego:

Robert potrzebował 50 PLN. Sprawdził więc stan konta i okazało się, że jest na nim 100 PLN. Nie ma problemu. A zatem zaplanował pobranie pieniędzy z konta.

Lecz wtedy zasnął!

I w tym momencie ja wkroczyłam do akcji. W czasie, gdy Robert spał, chciałam wyciągnąć 100 PLN. Sprawdziłam stan konta. Było na nim 100 PLN (ponieważ Robert zasnął i nie zakończył operacji pobierania pieniędzy). Uznałam zatem, że nie ma żadnego problemu i pobrałam pieniądze. Także i w tym momencie wszystko było w porządku. Ale wtedy obudził się Robert i dokonał swojej transakcji, no i stało się — nagle przekroczyliśmy limit! Nawet nie zdawał sobie sprawy, że zasnął, więc po prostu dokończył transakcję nie sprawdzając stanu konta. Musi nam pan pomóc, doktorze!”

Ale czy ten problem można rozwiązać? Czy los tej pary został już przesądzony? Nie możemy sprawić, by Robert nie zasypiał, ale czy możemy sprawić, że Monika nie będzie w stanie skorzystać z konta, dopóki jej mąż się nie obudzi?

Zastanówcie się proszę nad tym problemem, a my spotkamy się ponownie po przerwie reklamowej.



Monika i Robert — ofiary problemu „dwie osoby, jedno konto”.



Robert usnął po sprawdzeniu stanu konta, lecz przed pobraniem pieniędzy. Kiedy się obudził, bezwzględnie pobrał pieniądze bez ponownego sprawdzenia stanu konta.

Problem Moniki i Roberta w formie kodu

Poniższy przykład przedstawia, co może się zdarzyć, gdy *dwa* wątki (Monika i Robert) wspólnie używają *jednego* obiektu (konta bankowego).

Kod składa się z dwóch kas — KontoBankowe oraz MonikaIRobert. Klasa MonikaIRobert implementuje interfejs Runnable i reprezentuje działania, które wykonuje zarówno Monika, jak i Robert — sprawdzanie stanu konta oraz pobieranie gotówki. Jednak każdy z wątków jest usypany pomiędzy sprawdzeniem stanu konta a wypłaceniem pieniędzy.

Klasa MonikaIRobert ma składową konto typu KontoBankowe; składowa ta reprezentuje konto bankowe, które jest wspólnie używane przez oba wątki.

Poniżej przedstawiśmy sposób działania programu:

1 Utwórz jeden obiekt klasy MonikaIRobert.

Klasa MonikaIRobert implementuje interfejs Runnable (czyli określa zadanie, jakie należy wykonać). Poza tym oba wątki — zarówno Monika, jak i Robert — wykonują to samo zadanie (sprawdzają stan konta i wypłacają z niego gotówkę), i dlatego potrzebujemy tylko jednego obiektu tej klasy.

```
MonikaIRobert zadanie = new MonikaIRobert();
```

2 Utwórz dwa wątki, przekazując do obu ten sam obiekt Runnable (oczywiście będzie nim obiekt klasy MonikaIRobert).

```
Thread watek1 = new Thread(zadanie);
Thread watek2 = new Thread(zadanie);
```

3 Określ nazwy wątków i uruchom je.

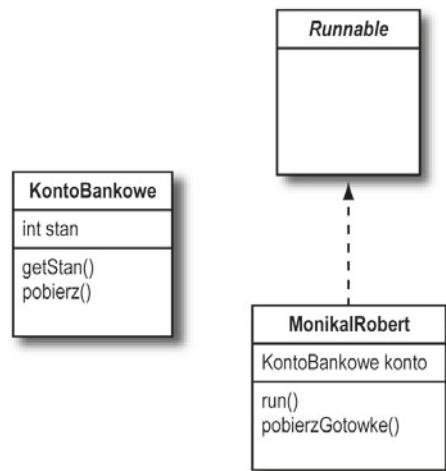
```
watek1.setName("Robert");
watek2.setName("Monika");
watek1.start();
watek2.start();
```

4 Obserwuj przebieg realizacji metody run() obu wątków

(czyli sprawdzanie stanu konta i pobieranie z niego gotówki).

Jeden z wątków reprezentuje Monikę, a drugi Roberta. Każdy z nich powtarza dwie czynności — sprawdza stan konta i pobiera z niego pieniądze. Gdyby tylko te operacje były bezpieczne!

```
if (konto.getStan() >= amount) {
    try {
        Thread.sleep(500);
    } catch(InterruptedException ex) {
        ex.printStackTrace();
    }
}
```



Metoda run() wykonuje dokładnie te same operacje, które wykonują Monika i Robert — sprawdza stan konta i, jeśli znajduje się na nim dostatecznie duża kwota, pobiera gotówkę.

To powinno stanowić wystarczające zabezpieczenie przed przekroczeniem limitu.

Jest tylko jeden problem... Monika i Robert zawsze zasypiają po sprawdzeniu stanu konta a przed zakończeniem operacji pobierania gotówki.

Przykład Moniki i Roberta

```

class KontoBankowe {
    private int stan = 100; ← Początkowo na koncie
    ← jest 100 PLN.

    public int getStan() {
        return stan;
    }

    public void pobierz(int kwota) {
        stan = stan - kwota;
    }
}

public class MonikaIRobert implements Runnable {
    private KontoBankowe konto = new KontoBankowe(); ← Będziemy używać tylko JEDNEGO
    obiektu MonikaIRobert. A to oznacza, że będzie dostępne tylko JEDNO konto bankowe (obiekt KontoBankowe). Oba wątki będą korzystać z tego jednego konta.

    public static void main (String [] args) {
        MonikaIRobert zadanie = new MonikaIRobert(); ← Tworzymy obiekt klasy implementującej interfejs Runnable (zadania wątków).
        Thread watek1 = new Thread(zadanie); ← Tworzymy dwa wątki, wykorzystując
        Thread watek2 = new Thread(zadanie); ← w każdym z nich to samo zadanie (obiekt Runnable). To oznacza, że oba wątki będą korzystać z tej samej składowej konta bankowego.

        watek1.setName("Robert");
        watek2.setName("Monika");
        watek1.start();
        watek2.start();
    }

    public void run() {
        for (int x = 0; x < 10; x++) {
            pobierzGotowke(10);
            if (konto.getStan() < 0) {
                System.out.println("Przekroczenie limitu!");
            }
        }
    }

    private void pobierzGotowke(int kwota) {
        if (konto.getStan() >= kwota) { ← W metodzie run() wątki próbują pobierać
            System.out.println(Thread.currentThread().getName() + " ma zamiar pobrać gotówkę."); ← gotówkę w pętli. Po pobraniu gotówki, stan konta jest ponownie sprawdzany, aby upewnić się, czy limit stanu konta nie zostanie przekroczony.

            try {
                System.out.println("Wątek " + Thread.currentThread().getName() + " zaraz zostanie uśpiony.");
                Thread.sleep(500);
            } catch(InterruptedException ex) {ex.printStackTrace();}
            System.out.println("Wątek " + Thread.currentThread().getName() + " obudził się.");
            konto.pobierz(kwota);
            System.out.println("Wątek " + Thread.currentThread().getName() + " zakończył operację.");
        }
        else {
            System.out.println("Przykro mi, brak środków dla wątku " + Thread.currentThread().getName());
        }
    }
}

```

W kodzie metody umieściliśmy kilka wywołań wyświetlających stosowne komunikaty, abyśmy wiedzieli, co się dzieje podczas jej realizacji.

Jesteś tutaj ▶ 531

Wyniki działania symulacji Moniki i Roberta

```
ca. Wiersz polecenia

Wątek Monika zaraz zostanie uśpiony.
Wątek Monika obudził się.
Wątek Monika zakończył operację.
Robert ma zamiar pobrać gotówkę.
Wątek Robert zaraz zostanie uśpiony.
Wątek Robert obudził się.
Wątek Robert zakończył operację.
Monika ma zamiar pobrać gotówkę.
Wątek Monika zaraz zostanie uśpiony.
Wątek Monika obudził się.
Wątek Monika zakończył operację.
Robert ma zamiar pobrać gotówkę.
Wątek Robert zaraz zostanie uśpiony.
Wątek Robert obudził się.
Wątek Robert zakończył operację.
Monika ma zamiar pobrać gotówkę.
Wątek Monika zaraz zostanie uśpiony.
Wątek Monika obudził się.
Wątek Monika zakończył operację.
Robert ma zamiar pobrać gotówkę.
Wątek Robert zaraz zostanie uśpiony.
Wątek Robert obudził się.
Wątek Robert zakończył operację.
Monika ma zamiar pobrać gotówkę.
Wątek Monika zaraz zostanie uśpiony.
Wątek Monika obudził się.
Wątek Monika zakończył operację.
Przekroczenie limitu!
Przykro mi, brak środków dla wątku Robert
Przekroczenie limitu!
Przykro mi, brak środków dla wątku Robert
Przekroczenie limitu!
Przykro mi, brak środków dla wątku Robert
Przekroczenie limitu!
Przykro mi, brak środków dla wątku Robert
Przekroczenie limitu!
Przykro mi, brak środków dla wątku Robert
Przekroczenie limitu!
Przykro mi, brak środków dla wątku Monika
Przekroczenie limitu!
Przykro mi, brak środków dla wątku Monika
Przekroczenie limitu!
Przykro mi, brak środków dla wątku Monika
Przekroczenie limitu!
Przykro mi, brak środków dla wątku Monika
Przekroczenie limitu!
```

Jak to się stało? →

Metoda pobierzGotowke() zawsze sprawdza stan konta przed wykonaniem operacji pobrania pieniędzy, jednak pomimo to doszło do przekroczenia limitu.

Oto jeden z możliwych scenariuszy:

Robert sprawdza stan konta, upewnia się, że jest na nim wystarczająco dużo pieniędzy, po czym zasypia.

W tym czasie Monika wkracza do akcji i sprawdza stan konta. Także ona dowiaduje się, że na koncie jest wystarczająco dużo środków. Monika nie ma pojęcia, że Robert zaraz się obudzi i dokończy rozpoczętą operację.

Monika zasypia.

Robert budzi się i kończy pobieranie pieniędzy.

Monika budzi się i kończy pobieranie pieniędzy. No i mamy problem! W chwili, gdy Monika już sprawdziła stan konta, lecz jeszcze zanim dokonała pobrania gotówki, obudził się Robert, który dokończył swoją operację pobierania.

Sprawdzenie stanu konta wykonane przez Monikę było nieważne, ponieważ Robert sprawdził stan konta wcześniej, lecz jeszcze nie zdążył zakończyć operacji pobierania.

Monika nie może mieć dostępu do konta aż do momentu, gdy Robert się obudzi i zakończy swoją transakcję. To samo zresztą dotyczy Roberta.

Monika i Robert muszą w jakiś sposób blokować dostęp do konta!

Blokada działa w sposób następujący:

- ① Z transakcją wykonywaną na koncie bankowym (w naszym przypadku jest nią sprawdzenie stanu konta i pobranie pieniędzy) jest skojarzona blokada (możesz ją sobie wyobrazić jako kłódkę). Istnieje tylko jeden klucz do tej blokady i do momentu, gdy ktoś spróbuje uzyskać dostęp do konta, jest on przechowywany razem z nią.



Kiedy nikt nie używa konta, transakja bankowa jest odblokowana.

- ② Kiedy Robert spróbuje skorzystać z konta bankowego (aby sprawdzić jego stan i pobrać pieniądze), zamyka blokadę, a klucz chowa do kieszeni. Teraz nikt inny nie może uzyskać dostępu do konta, gdyż nie ma niezbędnego klucza.



Kiedy Robert chce użyć konta, zamknięta blokadę i zabiera klucz.

- ③ Robert przez cały czas trwania transakcji trzyma klucz do blokady w kieszeni. Ponieważ jest to jedyny klucz, Monika nie może skorzystać z konta (ani z książeczki czekowej) aż do momentu, gdy Robert je odblokuje i odda klucz. Dzięki temu, nawet jeśli Robert zaśnie po sprawdzeniu stanu konta, ma gwarancję, że jego stan się nie zmieni, gdyż to on ma klucz!



Kiedy Robert skończy transakcję, zdejmuje blokadę i zwraca klucz. Teraz z klucza może skorzystać Monika (lub ponownie Robert).

Musimy wykonać metodę pobierzGotowke() jako operację atomową



Musimy mieć pewność, że gdy wątek rozpoczęte wykonywanie metody `pobierzGotowke()`, *będzie mógł ją dokończyć*, zanim zacznie być wykonywany inny wątek.

Innymi słowy, musimy mieć pewność, iż gdy wątek sprawdzi stan konta, jednocześnie uzyska gwarancję, że jego realizacja zostanie wznowiona, a on będzie miał szansę zakończyć operację pobierania pieniędzy, *zanim jakikolwiek inny wątek będzie mógł sprawdzić stan konta!*

Aby zmodyfikować metodę w taki sposób, by w danej chwili mógł ją wykonywać tylko jeden wątek, dodaj do niej słowo kluczowe **synchronized**.

Oto sposób, w jaki możesz ochronić swoje konto bankowe! Jednak blokowane nie jest samo konto, a jedynie metoda realizująca transakcję. W ten sposób jeden wątek może wykonać transakcję od początku do końca, nawet jeśli w jej trakcie zostanie uśpiony.

A zatem, jeśli nie blokujemy konta bankowego, to czym w zasadzie *jest* blokada? Czy jest metodą? Obiektem klasy implementującej interfejs `Runnable`? A może samym wątkiem?

Tym zagadnieniem zajmiemy się na następnej stronie. Z punktu widzenia tworzonego kodu wykorzystanie blokady jest bardzo proste — wystarczy dodać modyfikator `synchronized` do deklaracji metody:

```
private synchronized void pobierzGotowke(int kwota) {  
    if (konto.getStan() >= kwota) {  
        System.out.println(Thread.currentThread().getName() + " ma zamiar pobrać gotówkę.");  
        try {  
            System.out.println("Wątek " + Thread.currentThread().getName() + " zaraz zostanie uśpiony.");  
            Thread.sleep(500);  
        } catch(InterruptedException ex) {ex.printStackTrace();}  
        System.out.println("Wątek " + Thread.currentThread().getName() + " obudził się.");  
        konto.pobierz(kwota);  
        System.out.println("Wątek " + Thread.currentThread().getName() + " zakończył operację.");  
    } else {  
        System.out.println("Przykro mi, brak środków dla wątku " + Thread.currentThread().getName());  
    }  
}
```



Notatka dla osób z uzdolnieniami w dziedzinie fizyki: tak, cata przyjęta w Javie konwencja związana z wykorzystaniem słowa „atomowy” nie uwzględnia wszystkich zagadnień związanych z częstotliwościami subatomowymi. Zatem mówiąc „atomowy” w kontekście wątków lub transakcji, powinieneś myśleć raczej o Newtonie niż o Einsteinie. Hej, to nie *MY* wymyśliliśmy tą konwencję. Gdyby to od *NAS* zależało, to niemal do wszystkiego, co się wiąże z wątkami, zastosowalibyśmy zasadę nieoznaczości Heisenberga.



Słowo kluczowe synchronized oznacza, że wątek będzie potrzebował klucza, aby dostać się do synchronizowanego fragmentu kodu.

Aby chronić dane (takie jak konto bankowe), należy synchronizować metody, które na tych danych operują.

Stosowanie blokady obiektu

Każdy obiekt ma blokadę. W większości przypadków blokada ta nie jest używana i możesz sobie wyobrazić wirtualny kluczyk wiszący obok niej. Blokada obiektu zaczyna mieć znaczenie dopiero w sytuacji, gdy będą wykorzystywane metody synchronizowane. Jeśli obiekt ma jedną lub kilka metod synchronizowanych, **to wątek będzie mógł wykonać taką metodę, jeśli będzie w stanie uzyskać klucz do blokady obiektu.**

Blokady nie dotyczą *metod*, ograniczają one dostęp do *obiektów*. Jeśli obiekt ma dwie synchronizowane metody, nie oznacza to, że dwa wątki nie mogą wykonywać tej samej metody. Oznacza to jedynie, że dwa wątki nie mogą jednocześnie wykonywać *którejkolwiek* z tych synchronizowanych metod.

Zastanów się nad tym. Jeśli istnieje kilka metod, które potencjalnie mogą operować na składowych obiektu, to każdą z nich trzeba zabezpieczyć przy użyciu słowa kluczowego *synchronized*.

Zadaniem synchronizacji jest zabezpieczenie kluczowych danych obiektu. Pamiętaj jednak, że to nie same dane są synchronizowane, lecz metody, które na tych danych *operują*.

Co się zatem dzieje, gdy wątek wykonuje metody umieszczone na jego stosie wywołań (włącznie z metodą *run()*) i nagle natrafia na metodę synchronizowaną? Taki wątek orientuje się, że przed „wejściem” do metody musi uzyskać klucz do danego obiektu. Zatem szuka tego klucza (wszystkie te operacje są realizowane przez wirtualną maszynę Javy; nie istnieje żaden interfejs programistyczny pozwalający na operowanie na blokadach obiektów) i jeśli tylko klucz będzie dostępny, zabiera go i zaczyna wykonywać metodę.

Od tego momentu wątek dba o klucz, jakby to od niego zależało jego życie. Wątek nie odda klucza aż do momentu zakończenia metody. A zatem, dopóki wątek ma klucz, żaden inny wątek nie będzie w stanie wykonać jakiejkolwiek z synchronizowanych metod obiektu, a to dlatego, że klucz do obiektu nie jest dostępny.



Każdy obiekt w Javie ma blokadę. Do każdej blokady jest dołączony jeden klucz.

W większości przypadków blokada nie jest założona i nikt na nią nie zwraca uwagi.

Jednak jeśli obiekt posiada metody synchronizowane, to wątek może rozpoczęć wykonywanie jednej z tych metod wyłącznie w przypadku, gdy będzie dostępny klucz do blokady obiektu. Innymi słowy tylko wtedy, gdy inny wątek wcześniej nie wziął tego klucza.

Przerządzający problem „utraconej modyfikacji”

Oto kolejny, typowy przykład problemu związanego ze współbieżnością, pochodzący tym razem ze świata baz danych. Jest on ściśle związany z opowieścią o Monice i Robercie, jednak wykorzystamy go do przedstawienia kilku dodatkowych zagadnień.

Problem utraconej modyfikacji dotyczy następującego procesu:

Krok 1. Pobranie stanu konta

```
int i = stanKonta;
```

Krok 2. Powiększenie stanu konta o 1

```
stanKonta = i + 1;
```

Sztuczka niezbędna do zaprezentowania tego problemu polega na tym, aby zmusić komputer, by obie powyższe czynności zostały wykonane w dwóch etapach. W praktyce obie operacje wykonałbyś zapewne przy użyciu jednej instrukcji:

```
stanKonta++;
```

Niemniej jednak, wymuszając wykonanie całego zadania w *dwóch* etapach, wyraźnie pokażemy, na czym polega problem procesów nieatomowych. A zatem, wyobraź sobie, że zamiast trywialnego zadania — „pobierz wartość składowej `stanKonta` i powiększ ją o jeden” — wszystkie czynności realizowane przez metodę są znacznie bardziej złożone i nie da się ich wykonać przy użyciu jednej instrukcji.

W prezentacji problemu „utraconej modyfikacji” biorą udział dwa wątki, z których każdy stara się powiększyć stan konta.

```
class TestSynchro implements Runnable {  
  
    private int stanKonta;  
  
    public void run() {  
        for (int i = 0; i < 50; i++) { ←  
            inkrementuj();  
            System.out.println("Stan konta wynosi: " + stanKonta);  
        }  
    }  
  
    public void inkrementuj() {  
        int i = stanKonta; ←  
        stanKonta = i + 1; ←  
    }  
  
    public class TestSynchronizacji {  
        public static void main(String[] args) {  
            TestSynchro zadanie = new TestSynchro();  
            Thread watek1 = new Thread(zadanie);  
            Thread watek2 = new Thread(zadanie);  
            watekA.start();  
            watekB.start();  
        }  
    }  
}
```

W każdym wątku zawartość pętli jest wykonywana 50 razy, a podczas każdej iteracji inkrementowany jest stan konta.

A to jest najważniejsza część programu! Inkrementujemy stan konta, dodając wartość 1 do dowolnej wartości, jaką miała składowa `stanKonta` W CZASIE, GDY ODCZYTYWALISMY JEJ WARTOŚĆ (a nie dodając 1 do jej AKTUALNEJ wartości).

Wykonajmy ten przykładowy kod...

1 Przez pewien czas działa wątek A...



Zapisujemy wartość składowej `stanKonta` w zmiennej `i`.

Stan konta wynosi 0, zatem także wartość zmiennej `i` wynosi 0.

Przypisujemy składowej `stanKonta` wartość `i + 1`.

Aktualnie stan konta wynosi 1.

Zapisujemy wartość składowej `stanKonta` w zmiennej `i`.

Stan konta wynosi 1, zatem także wartość zmiennej `i` wynosi 1.

Przypisujemy składowej `stanKonta` wartość `i + 1`.

Aktualnie stan konta wynosi 2.

2 Przez pewien czas działa wątek B...



Zapisujemy wartość składowej `stanKonta` w zmiennej `i`.

Stan konta wynosi 2, zatem także wartość zmiennej `i` wynosi 2.

Przypisujemy składowej `stanKonta` wartość `i + 1`.

Aktualnie stan konta wynosi 3.

Zapisujemy wartość składowej `stanKonta` w zmiennej `i`.

Stan konta wynosi 3, zatem także wartość zmiennej `i` wynosi 3.

[W tym momencie wątek B jest przerywany i wraca do stanu wykonywalny, zanim zdąży zapisać wartość 4 w składowej `stanKonta`.]

3 Ponownie działa wątek A, zaczynając w miejscu, w którym został przerwany



Zapisujemy wartość składowej `stanKonta` w zmiennej `i`.

Stan konta wynosi 3, zatem także wartość zmiennej `i` wynosi 3.

Przypisujemy składowej `stanKonta` wartość `i + 1`.

Aktualnie stan konta wynosi 4.

Zapisujemy wartość składowej `stanKonta` w zmiennej `i`.

Stan konta wynosi 4, zatem także wartość zmiennej `i` wynosi 4.

Przypisujemy składowej `stanKonta` wartość `i + 1`.

Aktualnie stan konta wynosi 5.

4 Ponownie działa wątek B, a jego realizacja jest kontynuowana dokładnie od miejsca, w którym został przerwany



Przypisujemy składowej `stanKonta` wartość `i + 1`.

Aktualnie stan konta wynosi 4.

Ups!!

Wątek A przypisał składowej `stanKonta` wartość 5, jednak aktualnie zostało wznowiony wątek B, który przypisał składowej nową wartość, niwelując zmianę wprowadzoną przez wątek A. W rezultacie mogłoby to wyglądać tak, jak gdyby wątek A w ogóle tej zmiany nie wprowadził.

Straciliśmy ostatnią modyfikację wprowadzoną przez wątek A!
Wątek B wcześniej odczytał wartość składowej stanKonta, następnie został przerwany, a po „obudzeniu się” zamierza kontynuować działanie, jak gdyby w ogóle nic się nie stało.

Zadeklaruj metodę inkrementuj() jako metodę atomową. Synchronizuj ją!



Synchronizowanie metody inkrementuj() rozwiązuje problem „utraconej modyfikacji”, gdyż sprawia, że dwie tworzące ją instrukcje są wykonywane jako jeden niepodzielny fragment kodu.

```
public synchronized void inkrementuj() {  
    int i = stanKonta;  
    stanKonta = i + 1;  
}
```

Musimy zagwarantować, że gdy wątek zacznie realizować metodę, to wszystkie jej instrukcje zostaną wykonane (jako jeden atomowy proces), zanim jakikolwiek inny proces będzie mógł wykonać tę metodę.

Nie istnieją głupie pytania

P: Wygląda na to, że najlepszym rozwiązaniem będzie synchronizowanie wszystkich metod, aby zapewnić poprawne działanie wątków.

O: Bynajmniej nie jest to najlepszy pomysł. Korzyści, jakie daje synchronizacja, mają swoją cenę. Po pierwsze realizacja synchronizowanych metod jest obarczona pewnym narzutem czasowym. Innymi słowy, kiedy kod ma wykonać synchronizowaną metodę, jego efektywność działania spadnie (choć zazwyczaj nigdy tego nie będziesz w stanie zauważyc) w związku z koniecznością określenia, czy „dostępny jest klucz do blokady obiektu”.

Poza tym stosowanie metod synchronizowanych może pogorszyć efektywność działania programu, gdyż synchronizacja ogranicza możliwości działania współbieżnego. Innymi słowy, synchronizowana metoda zmusza wątki, aby „ustawiły się w kolejce i czekały na swoją kolej”. Być może w Twoim kodzie nie będzie to powodować żadnych problemów, jednak powinieneś wziąć to pod uwagę.

I ostatnia, choć jednocześnie najbardziej przerążająca sprawa — metody synchronizowane mogą doprowadzić do wystąpienia wzajemnej blokady (zajmiemy się tym na stronie 540).

Dobrą zasadą, jaką można stosować, jest synchronizowanie jak najmniejszej ilości metod — tylko tych, które powinny być synchronizowane. W rzeczywistości istnieje możliwość synchronizowania mniejszych bloków kodu niż całe metody.

Nie omawiamy tego zagadnienia w niniejszej książce, lecz można używać słowa kluczowego `synchronized` w celu synchronizowania kodu na poziomie pojedynczych instrukcji, a nie całych metod.

Metoda `doRoboty()` nie musi być synchronizowana, zatem nie synchronizujemy jej.

```
public void doRoboty() {  
    zrobCos();
```

```
synchronized(this) {  
    kluczoweZadanie();  
    inneKluczoweZadanie();  
}
```

}

Jedynie te dwa wywołania zostały zgrupowane w jeden atomowy blok kodu. W przypadku umieszczenia słowa kluczowego `synchronized` wewnątrz kodu, a nie w deklaracji metody, trzeba dodać do niego argument określający obiekt, który będzie używany do synchronizacji dostępu do bloku atomowego.

Choć istnieją także i inne możliwości, to jednak niemal zawsze synchronizacja będzie realizowana na podstawie bieżącego obiektu (`this`). Ten sam obiekt jest wykorzystywany do synchronizowania dostępu do całych metod.

① Wątek A działa przez chwilę.



Podejmujemy próbę wykonania metody `inkrementuj()`.

Metoda jest synchronizowana, zatem **pobieramy klucz** do blokady obiektu.

Zapisujemy wartość składowej `stanKonta` w zmiennej `i`.

Stan konta wynosi 0, zatem także wartość zmiennej `i` wynosi 0.

Przypisujemy składowej `stanKonta` wartość `i + 1`.

Nowy stan konta wynosi 1.

Zwracamy klucz do blokady obiektu (gdyż zakończyliśmy wykonywanie metody `inkrementuj()`).

Ponownie wykonujemy metodę `inkrementuj()` i **pobieramy klucz** do blokady obiektu.

Zapisujemy wartość składowej `stanKonta` w zmiennej `i`.

Stan konta wynosi 1, zatem także wartość zmiennej `i` wynosi 1.

[W tym momencie stan wątku A zmienia się na wykonywalny, jednak wątek ten wciąż posiada klucz do obiektu, gdyż realizacja synchronizowanej metody `inkrementuj()` jeszcze nie została zakończona.]

② Wątek B zostaje wybrany i staje się aktualnie wykonywanym wątkiem.



Próbujemy uruchomić metodę `inkrementuj()`.

Metoda jest synchronizowana, zatem potrzebny nam jest klucz do blokady obiektu.

Klucz nie jest dostępny.

[Wątek B jest odsyłany do poczekalni wątków oczekujących na klucz do blokady obiektu.]

③ Ponownie zaczyna być wykonywany wątek A, a jego działanie jest wznowiane w miejscu, w którym wcześniej zostało przerwane (pamiętaj, że ten wątek wciąż posiada klucz).



Przypisujemy składowej `stanKonta` wartość `i + 1`.

Nowy stan konta wynosi 2.

Zwracamy klucz do blokady obiektu.

[W tym momencie stan wątku A zmienia się na wykonywalny, a ponieważ metoda `inkrementuj()` została wykonana, zatem wątek nie ma już klucza do blokady obiektu.]

④ Wznawiana jest realizacja wątku B.



Próbujemy uruchomić metodę `inkrementuj()`.

Metoda jest synchronizowana, zatem potrzebny nam jest klucz do blokady obiektu.

Tym razem klucz jest dostępny, a zatem go pobieramy.

Zapisujemy wartość składowej `stanKonta` w zmiennej `i`.

[dalsza część programu wykonywana jest tak samo...]

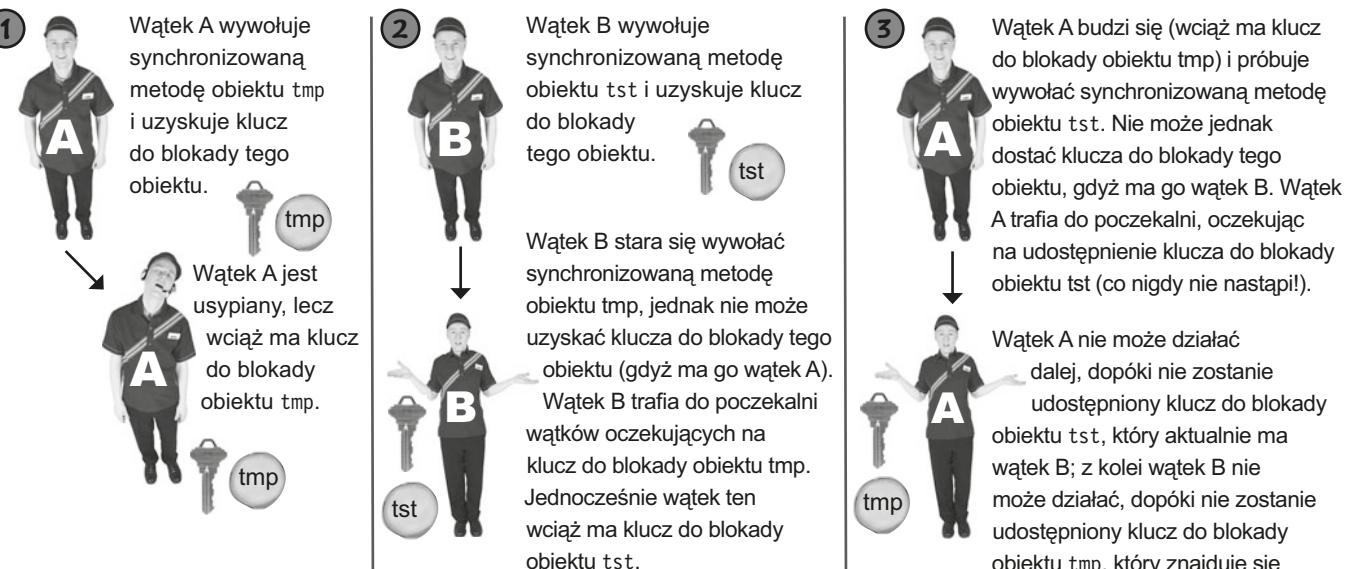
Mroczna strona synchronizacji

Trzeba bardzo uważać, gdzie używamy synchronizowanego kodu, gdyż nic nie jest w stanie tak łatwo rzucić naszego programu na kolana, jak wzajemna blokada wątków. Wzajemna blokada wątków występuje w sytuacji, gdy istnieją dwa wątki, z których każdy ma klucz do blokady czegoś, czego potrzebuje ten drugi. Z takiej sytuacji nie ma już wyjścia. A zatem oba wątki usiądą i będą czekać. I czekać... I czekać...

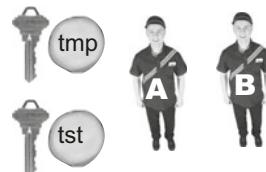
Jeśli znasz się na bazach danych lub innych aplikacjach serwerowych, to na pewno rozpoznasz ten problem; bazy danych mają zazwyczaj mechanizmy blokowania danych, które nieco przypominają synchronizację. Jednak prawdziwe systemy zarządzania transakcjami czasami są w stanie poradzić sobie ze wzajemną blokadą. Na przykład mogą one uznawać, że wystąpiła wzajemna blokada w przypadku, gdy realizacja transakcji trwa zbyt długo. Jednak w odróżnieniu od Javy systemy takie są w stanie „odtworzyć transakcję”, czyli przywrócić stan sprzed rozpoczęcia transakcji (czyli grupy operacji atomowych).

Java nie jest wyposażona w żadne mechanizmy pozwalające na rozwiązywanie wzajemnej blokady. Co więcej, nawet nie będzie *wiedzieć*, że wzajemna blokada wystąpiła. A zatem to Ty musisz uważnie zaprojektować swój program. Jeśli okaże się, że Twój program w dużym stopniu korzysta z wielowątkowości, możesz uznać za wskazane, by przeczytać książkę *Java Threads* Scotta Oaksa i Henry'ego Wonga, w której znajdziesz porady dotyczące sposobów projektowania programów w taki sposób, by unikać występowania wzajemnej blokady. Jedna z podstawowych porad dotyczy zwracania bacznej uwagi na kolejność uruchamiania wątków.

Prosty przykład powstawiania wzajemnej blokady:



Do spowodowania wzajemnej blokady wystarczą dwa obiekty i dwa wątki.





CELNE SPOSTRZEŻENIA

- Statyczna metoda `Thread.sleep()` wymusza przerwanie wykonywania wątku, przy czym długość przerwy w jego realizacji nie może być krótsza od czasu podanego w wywołaniu metody. Wywołanie `Thread.sleep(2000)` „usypia” wątek na co najmniej 2000 milisekund.
- Metoda `sleep()` zgłasza sprawdzany wyjątek (`InterruptedException`), a zatem wszystkie jej wywołania muszą być umieszczane wewnątrz bloków `try-catch`, ewentualnie należy deklarować ten wyjątek.
- Wywoływanie metody `sleep()` może pomóc w uzyskaniu większej pewności, że wszystkie wątki będą realizowane. Nie ma jednak żadnej pewności, że po „obudzeniu” wątek trafi na sam koniec kolejki wątków oczekujących na wykonanie. Równie dobrze może trafić, na przykład, na sam jej początek. W większości przypadków wywołania metody `sleep()` umieszczone w odpowiednich miejscach i przerywające działanie wątku na odpowiednio dobrany czas w zupełności wystarczają, by zapewnić poprawną, naprzemienną realizację wszystkich wątków.
- Przy użyciu metody `setName()` można określić nazwy wątków. Wszystkim wątkom nadawane są nazwy domyślne, jednak jawne nadanie każdemu z nich nazwy może Ci pomóc w śledzeniu ich działania; zwłaszcza jeśli używasz do tego celu metody `print()`.
- Stosowanie wielu wątków może przysporzyć poważnych problemów, zwłaszcza jeśli dwa lub więcej wątków mają dostęp do tego samego obiektu na stercie.
- W przypadku, gdy dwa lub większa ilość wątków operuje na jednym obiekcie, może dojść do zniekształcenia danych. Dotyczy to zwłaszcza tych sytuacji, gdy realizacja jednego z wątków zostaje przerwana w trakcie modyfikowania kluczowych danych określających stan obiektu.
- Aby obiekt mógł być bezpiecznie wykorzystywany przez wiele wątków, trzeba określić, które instrukcje należy traktować jako proces atomowy. Innymi słowy, należy ustalić, które z metod będą musiały zostać wykonane w całości, zanim inny wątek będzie mógł wywołać tę samą metodę tego samego obiektu.
- Aby uniemożliwić jednocześnie wykonywanie tej samej metody przez dwa różne wątki, dodaj do jej deklaracji słowo kluczowe **synchronized**.
- Każdy obiekt ma blokadę oraz jeden klucz do niej. Zazwyczaj nie będziemy zwracać uwagi na tę blokadę — blokady zaczynają mieć znaczenie wyłącznie w przypadku, gdy obiekt ma synchronizowane metody.
- Kiedy wątek próbuje wywołać metodę synchronizowaną, musi najpierw uzyskać klucz do blokady obiektu (przy czym chodzi o obiekt, którego metodę wątek chce wywołać). Jeśli klucz nie jest dostępny (gdyż wcześniej dostał go inny wątek), wątek próbujący wywołać metodę trafia do swoistej „poczekalni”, gdzie będzie czekać aż do chwili, gdy klucz zostanie zwrócony.
- Nawet jeśli obiekt ma kilka synchronizowanych metod, to i tak istnieje tylko jeden klucz do jego blokady. Kiedy wątek wywołał i zaczął realizować synchronizowaną metodę pewnego obiektu, to żaden inny wątek nie może wywołać jakichkolwiek innych synchronizowanych metod tego samego obiektu. Ograniczenie to pozwala na ochronę danych poprzez synchronizację metod, które nimi manipulują.

Nowa i poprawiona wersja programu ProstyKlientPogawedek

Całe wieki temu — gdzieś na początku tego rozdziału — napisaliśmy program ProstyKlientPogawedek, który mógł wysyłać wiadomości na serwer, lecz nie był w stanie niczego odbierać. Pamiętasz go może? To właśnie on doprowadził nas do tego całego zagadnienia wątków — potrzebowaliśmy bowiem sposobu na jednoczesne wykonywanie dwóch operacji. Konkretnie rzecz biorąc, musielibyśmy jednocześnie wysyłać wiadomość *na* serwer (wykorzystując przy tym komponenty graficznego interfejsu użytkownika) oraz odbierać wiadomości przesyłane z serwera i wyświetlać je w wielowierszowym polu tekstowym.

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ProstyKlientPogawedek {

    JTextArea odebraneWiadomosci;
    JTextField wiadomosc;
    BufferedReader czytelnik;
    PrintWriter pisarz;
    Socket gniazdo;

    public static void main(String[] args) {
        ProstyKlientPogawedek klient = new ProstyKlientPogawedek();
        klient.doDziela();
    }

    public void doDziela()
    {
        JFrame ramka = new JFrame("Śmiesznie prosty klient pogawędek");
        JPanel panelGlowny = new JPanel();

        odebraneWiadomosci = new JTextArea(15,50);
        odebraneWiadomosci.setLineWrap(true);
        odebraneWiadomosci.setWrapStyleWord(true);
        odebraneWiadomosci.setEditable(false);

        JScrollPane przewijanie = new JScrollPane(odebraneWiadomosci);
        przewijanie.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        przewijanie.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        wiadomosc = new JTextField(20);

        JButton przyciskWyslij = new JButton("Wyślij");
        przyciskWyslij.addActionListener(new PrzyciskWyslijListener());

        panelGlowny.add(przewijanie);
        panelGlowny.add(wiadomosc);
        panelGlowny.add(przyciskWyslij);
        konfigurujKomunikacje();

        Thread watekOdbiorcy = new Thread(new OdbiorcaKomunikatow());
        watekOdbiorcy.start();

        ramka.getContentPane().add(BorderLayout.CENTER, panelGlowny);
        ramka.setSize(400,500);
        ramka.setVisible(true);
    } // koniec metody
}
```

Tak, gdzieś jest koniec tego rozdziału... gdzieś, ale jeszcze nie tu.

To jest w przeważającej większości kod, który widziałeś już wcześniej — odpowiada on za utworzenie graficznego interfejsu użytkownika. Nie ma tu nic szczególnego za wyjątkiem wyróżnionego fragmentu, w którym uruchamiamy wątek „odbiorcy”.

Uruchamiamy nowy wątek, przekazując do niego obiekt klasy wewnętrznej (określający zadanie, jakie wątek ma wykonać). Zadaniem wątku jest odczytywanie wiadomości ze strumienia wejściowego, do którego trafiają dane przesypane z serwera, i wyświetlanie ich w wielowierszowym polu tekstowym.

```

private void konfigurujKomunikacje() {
    try {
        gniazdo = new Socket("127.0.0.1", 5000);
        InputStreamReader czytelnikStrm = new InputStreamReader(gniazdo.getInputStream());
        czytelnik = new BufferedReader(czytelnikStrm);
        pisarz = new PrintWriter(gniazdo.getOutputStream());
        System.out.println("obsługa sieci przygotowana");
    } catch(IOException ex) {
        ex.printStackTrace();
    }
} // koniec metody

```

Do pobrania strumienia wejściowego i wyjściowego używamy gniazda. Strumienia wyjściowego używaliśmy już wcześniej do wysyłania wiadomości na serwer, jednak teraz używamy także strumienia wejściowego, dzięki któremu możemy odbierać wiadomości rozsyłane przez serwer.

```

public class PrzyciskWyslijListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
        try {
            pisarz.println(wiadomosc.getText());
            pisarz.flush();

        } catch(Exception ex) {
            ex.printStackTrace();
        }
        wiadomosc.setText("");
        wiadomosc.requestFocus();
    }
} // koniec klasy wewnętrznej

```

W tym fragmencie kodu nie ma niczego nowego. Kiedy użytkownik kliknie przycisk „Wyślij”, metoda wysyła zawartość pola tekstowego na serwer.

```

public class OdbiorcaKomunikatow implements Runnable {
    public void run() {
        String wiadom;
        try {
            while ((wiadom = czytelnik.readLine()) != null) {
                System.out.println("Odczytano: " + wiadom);
                odebraneWiadomosci.append(wiadom + "\n");
            } // koniec while
        } catch(Exception ex) {ex.printStackTrace();}
    } // koniec metody run()
} // koniec klasy wewnętrznej

```

To właśnie jest zadanie realizowane przez wątek!!! Wewnątrz metody run() wykonywana jest pętla (wykonywana do momentu, gdy dana przestana przez serwer będzie mieć wartość null), która po wierszu odczytuje wiadomości przestane z serwera i wyświetla je w wielowierszowym polu tekstowym (dodając do każdej linii znak nowego wiersza).

Kod serwera pogawędek



Kod gotowy do użycia

Naprawdę prosty serwer pogawędek

Poniższy program serwera może współpracować z obydwoma wersjami klienta. Przyznajemy się do wszystkich zastrzeżeń, jakie można mieć do tego kodu. Aby w jak największym stopniu ograniczyć ten program i wyposażyć go jedynie w zestaw absolutnie niezbędnych możliwości, usunęliśmy z niego wiele fragmentów, które byłyby potrzebne w normalnym serwerze, nadającym się do praktycznego zastosowania. Innymi słowy, program działa, choć można w nim wywołać awarię przynajmniej na sto różnych sposobów. Jeśli będziesz chciał wykonać naprawdę trudne ćwiczenie „Zaostrz ołówek”, to po zakończeniu lektury całej książki zajrzyj do tego kodu i spróbuj sprawić, aby był bardziej solidny i niezawodny.

Innym ćwiczeniem „Zaostrz ołówek”, które możesz wykonać (i to już teraz), jest podanie własnych przypisów do tego kodu. Znacznie lepiej zrozumiesz działanie tego programu, jeśli sam spróbujesz określić, co się w nim dzieje. Pamiętaj jednak, że to „Kod gotowy do użycia”, zatem prawdę mówiąc, w ogóle nie musisz rozumieć, co się w nim dzieje. Przedstawiamy go tylko po to, aby umożliwić przetestowanie zaprezentowanych wcześniej klientów pogawędek.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class BardzoProstySerwerPogawedek {

    ArrayList strumieniewyjsciowe;

    public class ObslugaKlientow implements Runnable {
        BufferedReader czytelnik;
        Socket gniazdo;

        public ObslugaKlientow(Socket clientSocket) {
            try {
                gniazdo = clientSocket;
                InputStreamReader isReader = new InputStreamReader(gniazdo.getInputStream());
                czytelnik = new BufferedReader(isReader);

            } catch(Exception ex) {ex.printStackTrace();}
        } // koniec konstruktora

        public void run() {
            String wiadomosc;
            try {
                while ((wiadomosc = czytelnik.readLine()) != null) {
                    System.out.println("Odczytano: " + wiadomosc);
                    rozeslijDowszystkich(wiadomosc);

                } // koniec pętli
            } catch(Exception ex) {ex.printStackTrace();}
        } // koniec metody
    } // koniec klasy wewnętrznej
```

**Aby uruchomić klienta pogawędek, będziesz potrzebował dwóch okien wiersza poleceń.
Najpierw w pierwszym oknie wiersza poleceń uruchom przedstawiony tu program serwera, a następnie, w drugim oknie, uruchom klienta.**

```
public static void main (String[] args) {
    new BardzoProstySerwerPogawedek().doRoboty();
}

public void doRoboty() {
    strumieniewyjsciowe = new ArrayList();
    try {
        ServerSocket serverSock = new ServerSocket(5000);

        while(true) {
            Socket gniazdoKlienta = serverSock.accept();
            PrintWriter pisarz = new PrintWriter(gniazdoKlienta.getOutputStream());
            strumieniewyjsciowe.add(pisarz);

            Thread t = new Thread(new ObslugaKlientow(gniazdoKlienta));
            t.start();
            System.out.println("mamy połaczenie");
        }
    } catch(Exception ex) {
        ex.printStackTrace ();
    }
} // koniec metody

public void rozeslijDowszystkich(String message) {
    Iterator it = strumieniewyjsciowe.iterator();
    while(it.hasNext()) {
        try {
            PrintWriter pisarz = (PrintWriter) it.next();
            pisarz.println(message);
            pisarz.flush();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    } // koniec pętli
} // koniec metody
} // koniec klasy
```

Pytania dotyczące synchronizacji

I Nie istnieja
głupie pytania

P: A jak wyglądają możliwości ochrony stanu składowych statycznych? Czy można używać synchronizacji w przypadku, gdy korzystamy z metod statycznych modyfikujących stan składowych statycznych?

O: Tak! Pamiętaj, że do wywoływanego metod statycznych używamy klas, a nie konkretnych obiektów. Możesz się zatem zastanawiać, jaka blokada będzie sprawdzana w przypadku wywoływanego metody statycznej. Przecież może się zdarzyć, że w chwili wywoływanego metody statycznej nie będzie nawet *istniał* żaden obiekt tej klasy. Na szczęście, nie tylko każdy *obiekt* ma swoją blokadę, lecz także każda wczytana *klasa*. A to oznacza, że w przypadku, gdy program używa trzech obiektów klasy *Pies*, będą istniały cztery blokady skojarzone z tą klasą i jej obiektami. Trzy spośród tych blokad będą skojarzone z obiektami, a czwarta z samą klasą *Pies*. W przypadku synchronizowania metod statycznych Java wykorzystuje blokadę skojarzoną z samą klasą. A zatem, jeśli w jednej klasie zostaną utworzone dwie synchronizowane metody statyczne, to aby wywołać którykolwiek z nich, wątek będzie musiał zdobyć klucz do blokady klasy.

P: A co z priorytetami wątków? Słyszałem, że jest to sposób pozwalający na kontrolę mechanizmu zarządzającego wykonywaniem wątków.

O: Określanie priorytetów wątków może się okazać pomocne podczas prób wpłynięcia na sposób działania mechanizmu zarządzającego wątkami. Niemniej jednak nawet priorytety nie dają nam żadnej gwarancji. Priorytety wątków to wartości liczbowe, które informują mechanizm zarządzający (jeśli w ogóle zwraca na to uwagę), jakie znaczenie ma dla Ciebie dany wątek. Ogólnie rzecz biorąc, mechanizm zarządzający przerwie realizację aktualnie wykonywanego wątku, jeśli jakikolwiek wątek o wyższym priorytecie stanie się „uruchamialny”. Ale... jeszcze raz powtórz to razem z nami: „nie ma żadnych gwarancji”. Zalecamy, abyś używał priorytetów wątków tylko w przypadkach, gdy chcesz wpłynąć na efektywność działania programu, natomiast nigdy nie powinna od nich zależeć poprawność jego działania.

P: A czy nie można by po prostu synchronizować wszystkich metod ustawiających i zwracających operujących na danych, które chcemy ochroniać? Na przykład dlaczego nie synchronizowaliśmy metod `getStan()` oraz `pobierz()` klasy `KontoBankowe`, zamiast metody `pobierzGotowke()` zdefiniowanej w klasie implementującej interfejs `Runnable`?

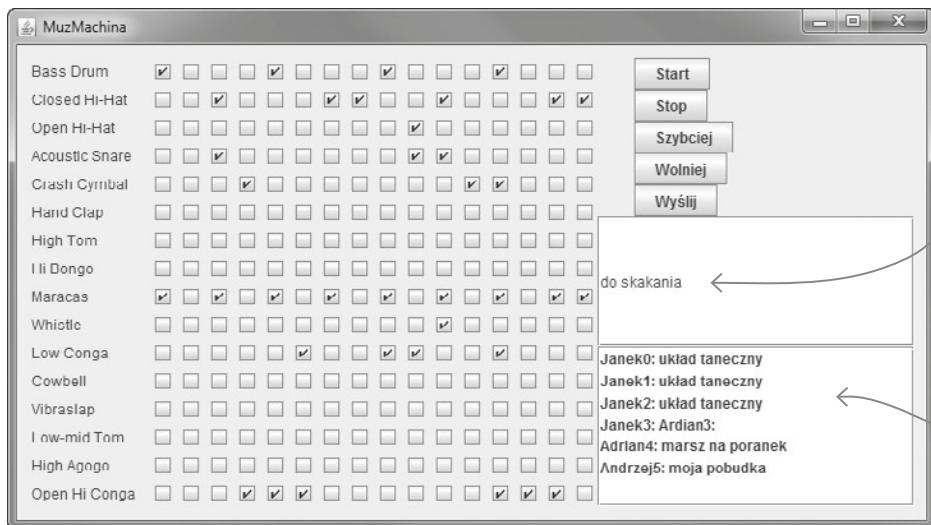
O: W zasadzie to powinniśmy synchronizować właśnie te metody, aby nie dać innym wątkom możliwości korzystania z nich na inne sposoby. Nie przejmowaliśmy się tym problemem, gdyż w naszym przykładzie żaden inny kod nie operował na koncie.

Niemniej jednak synchronizowanie metod zwracających i ustawiających (czyli w naszym przypadku, metod `getStan()` oraz `pobierz()`) nie wystarcza. Pamiętaj, że celem synchronizacji jest stworzenie bloku kodu wykonywanego w sposób ATOMOWY. Innymi słowy nie interesują nas pojedyncze metody, lecz metody **wykonujące więcej niż jedną operację!** Zastanów się nad tym. Gdyby metoda `pobierzGotowke()` nie była synchronizowana, to Robert sprawdziłby stan konta (wywołując w tym celu synchronizowaną metodę `getStan()`), natychmiast zakończył wykonywanie tej metody i wrócił klucz do blokady konta!

Oczywiście zaraz po obudzeniu się Robert znowu uzyskałby klucz do blokady konta, aby wywołać synchronizowaną metodę `pobierz()`, jednak takie rozwiązanie przysporzyłoby nam tych samych problemów, które mieliśmy przed zastosowaniem synchronizacji! Robert mógłby sprawdzić stan konta i zasnąć, a w czasie jego snu Monika mogłaby sprawdzić stan konta i dokonać wypłaty, zanim Robert uzyskałby możliwość zakończenia swojej transakcji.

A zatem synchronizowanie wszystkich metod zapewniających dostęp do danych jest zapewne dobrym rozwiązaniem zabezpieczającym przed niepożądanymi skutkami działania innych wątków. Niemniej jednak i tak trzeba będzie synchronizować metody zawierające instrukcje, które muszą być wykonywane jako jeden atomowy blok kodu.

Kod od kuchni



Kiedy klikniesz przycisk „Wyślij”, Twój wiadomość „Wyślij”, Twoja wiadomość jest wysyłana do innych uczestników pogawędki wraz z aktualną kompozycją.

Odebrane wiadomości wystawane przez innych uczestników pogawędki. Kliknij wiadomość, aby wczytać i odtworzyć utwór przestany wraz z nią.

Oto ostatnia wersja naszej aplikacji muzycznej!

Współpracuje ona z prostym serwerem muzycznym, aby można wysyłać i odbierać wiadomości i kompozycje przesypane przez innych uczestników pogawędki.

Kod serwera jest naprawdę długie, dlatego też jego pełna wersja została zamieszczona w dodatku A.

Ćwiczenia: Magnesiki z kodem



Ćwiczenie



Magnesiki z kodem

Działający program Javy został podzielony na fragmenty, zapisany na małych magnesach, które przyczepiono do lodówki. Czy jesteś w stanie złożyć magnesiki przedstawione na następnej stronie w jeden działający program, który wygeneruje wyniki przedstawione poniżej? Niektóre nawiasy klamrowe spadły na podłogę i były zbyt małe, aby można je było podnieść, dlatego w razie potrzeby możesz je dodawać!

```
public class TestWatkow {
```

```
class Sumator {
```

```
class WatekPierwszy {
```

```
class WatekDrugi {
```

Pytanie dodatkowe: Jak myślisz dlaczego zastosowaliśmy modyfikatory, które umieściliśmy w klasie Sumator?

```
Wiersz polecenia
T:>java TestWatkow
Pierwszy - 97099
Drugi - 97099
```

Ciąg dalszy magnesików z kodem...

```
Thread pierwszy = new Thread(w1);
```

```
} catch (InterruptedException ex) {}
```

```
Sumator s = Sumator.getSumator();
```

```
Thread drugi = new Thread(w2);
```

```
public static Sumator getSumator() {
```

```
private int licznik = 0;
```

```
s.aktualizujLicznik(1);
```

```
for (int x=0; x < 99; x++) {
```

```
public int getLicznik() {
```

```
public void aktualizujLicznik(int wart) {
```

```
for (int x=0; x < 98; x++) {
```

```
drugi.start();
```

```
try {
```

```
public void run() {
```

```
private Sumator() {}
```

```
Sumator s = Sumator.getSumator();
```

```
System.out.println("Drugi - " + s.getLicznik());
```

```
WatekDrugi w2 = new WatekDrugi();
```

```
try {
```

```
return licznik; licznik += wart;
```

```
implements Runnable {
```

```
pierwszy.start();
```

```
Thread.sleep(50);
```

```
} catch (InterruptedException ex) {}
```

```
private static Sumator s = new Sumator();
```

```
public void run() {
```

```
Thread.sleep(50);
```

```
implements Runnable {
```

```
s.aktualizujLicznik(1000);
```

```
return s;
```

```
System.out.println("Pierwszy - " + s.getLicznik());
```

```
WatekPierwszy w1 = new WatekPierwszy();
```

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

Rozwiązania ćwiczeń

```
public class TestWatkov {  
    public static void main(String[] args) {  
        WatekPierwszy w1 = new WatekPierwszy();  
        WatekDrugi w2 = new WatekDrugi();  
        Thread pierwszy = new Thread(w1);  
        Thread drugi = new Thread(w2);  
        pierwszy.start();  
        drugi.start();  
    }  
}  
  
class Sumator {  
    private static Sumator s = new Sumator();  
  
    private int licznik = 0;  
  
    private Sumator() {} ← Prywatny konstruktor.  
  
    public static Sumator getSumator() {  
        return s;  
    }  
  
    public void aktualizujLiczni(int wart) {  
        licznik += wart;  
    }  
  
    public int getLiczni() {  
        return licznik;  
    }  
}  
  
class WatekPierwszy implements Runnable {  
    Sumator s = Sumator.getSumator();  
    public void run() {  
        for (int x=0; x < 98; x++) {  
            s.aktualizujLiczni(1000);  
            try {  
                Thread.sleep(50);  
            } catch (InterruptedException ex) {}  
        }  
        System.out.println("Pierwszy - " +  
            s.getLiczni());  
    }  
}
```

Tworzymy obiekt Sumator
i zapisujemy go
w składowej statycznej.

Rozwiązania ćwiczeń

Wątki dwóch różnych klas aktualizują obiekt trzeciej klasy, gdyżoba wątki operują na statycznym obiekcie klasy Sumator. Poniższy wiersz kodu:

```
private static Sumator s = new Sumator();  
  
tworzy statyczną składową Sumator (pamiętaj — oznacza to, że w klasie istnieje tylko jedna taka składowa), natomiast zastosowanie prywatnego konstruktora oznacza, że nikt inny nie będzie w stanie utworzyć obiektu Sumator. Te dwie techniki (czyli prywatny konstruktor oraz statyczna metoda odczytująca) zastosowane razem tworzą tak zwany „Singleton” — wzorzec projektowy wykorzystywany do ograniczania ilości obiektów, które mogą istnieć w aplikacji. (Zazwyczaj może istnieć tylko jeden taki obiekt, stąd też pochodzi nazwa wzorca projektowego; niemniej jednak narzucone ograniczenia mogą być dowolne).
```

1) Angielskie słowo „singleton” oznacza osobę lub rzecz występującą pojedynczo.

```
class WatekDrugi implements Runnable {  
    Sumator s = Sumator.getSumator();  
    public void run() {  
        for (int x=0; x < 99; x++) {  
            s.aktualizujLiczni(1);  
            try {  
                Thread.sleep(50);  
            } catch (InterruptedException ex) {}  
        }  
        System.out.println("Drugi - " +  
            s.getLiczni());  
    }  
}
```



Zagadka na pięć minut



Problemy ze śluzami powietrznymi

Kiedy Sara dołączyła do zebrania pokładowego zespołu programistów poświęconego analizie projektu spojrzała przez iluminator na wschód słońca nad Oceanem Indyjskim. Choć sala konferencyjna statku mogła wywołać klaustrofobię, to jednak widok powiększającego się biało-niebieskiego półksiężyca rozświetlającego noc panującą na pobliskiej planecie, przepełnił Sarę respektem i podziwem.

Spotkanie tego ranka miało dotyczyć kontroli systemu śluz powietrznych stacji orbitalnej. Wraz ze zbliżającym się końcem ostatecznego etapu konstrukcji stacji planowana ilość wyjść w przestrzeń kosmiczną miała drastycznie wzrosnąć, a ruch w śluzach powietrznych — zarówno do wnętrza jak i na zewnątrz statku — miał być bardzo wysoki. — Dzień dobry, Saro — powiedział Tomek — Twoje wyczucie czasu jest po prostu niesamowite — właśnie mieliśmy zacząć szczegółowy przegląd projektu.

— Jak wszyscy wiecie — powiedział Tomek — każda śluza powietrzna jest wyposażona we wzmacnione terminale graficzne, zarówno po jej zewnętrznych, jak i wewnętrznych stronach. Za każdym razem,

gdy ktoś będzie opuszczał stację lub wchodził do niej, będzie musiał skorzystać z tych terminali, aby zapoczątkować sekwencje użycia śluzy. Sara kiwnęła głową i odezwała się:
— Tomku, czy możesz nam powiedzieć, jakie są sekwencje metod wykonywane podczas operacji wejścia oraz opuszczenia stacji?. Tomek odparchnął się od blatu i poszybował w kierunku tablicy. — Oto pseudokod sekwencji metod dla operacji opuszczenia stacji — powiedział i szybko napisał kilka wywołań:

```
sekwencja0obslugiSluzyPodczas0puszczaniaStacji()
    weryfikujStatusPortalu();
    wyrownajCisnienieWSluzie();
    otworzWewnetrzneDrzwiSluzy();
    potwierdz0becnoscPasazera();
    zamknijWewnetrzneDrzwiSluzy();
    dekomprezujSluze();
    otworzZewnetrzneDrzwiSluzy();
    potwierdz0proznienieSluzy();
    zamknijZewnetrzneDrzwiSluzy();
```

— Aby zapewnić, że sekwencja nie zostanie przerwana, synchronizowaliśmy wszystkie metody wchodzące w skład sekwencji — wyjaśnił Tomek — W końcu nie chcielibyśmy, aby kosmonauta wracający do stacji mógł otworzyć śluzę, gdy wewnętrz niej jest ktoś inny bez kombinezonu.

Gdy Tomek ścierał tablicę, wszyscy się roześmiali, jednak Sarze coś nie dawało spokoju. W końcu, kiedy Tomek już zaczął pisać na tablicy pseudokod sekwencji wejścia, Sara zrozumiała przyczynę swojego wewnętrznego niepokoju. — Chwileczkę Tomku! — krzyknęła. — Myślę, że mamy poważny błąd w projekcie naszej sekwencji wyjścia — musimy wrócić do pracy i przemyśleć sekwencję jeszcze raz, to może mieć ogromne znaczenie!

Dlaczego Sara przerwała spotkanie? Co podejrzewała?



O czym wiedziała Sara?

Sara uświadomiła sobie, że w celu zapewnienia nieprzerwanego wykonania sekwencji opuszczenia stacji cała metoda `sekwencja0bslugiSluzyPodczas0puszczaniaStacji()` musi być synchronizowana. Gdyby projekt pozostał w przedstawionej postaci, to wracający kosmonauta mógłby przerwać sekwencję opuszczenia stacji! Oczywiście nie można by przerwać wątku realizującego sekwencję opuszczenia podczas wykonywania którejkolwiek z metod niższego poziomu, jednak *istniałaby możliwość* przerwania go pomiędzy wywołaniami tych metod. Sara wiedziała, że cała sekwencja powinna być wykonywana jako jedna atomowa operacja oraz że w celu uniknięcia możliwości przerwania sekwencji konieczne jest synchronizowanie całej metody `sekwencja0bslugiSluzyPodczas0puszczaniaStacji()`.

16. Kolekcje i typy ogólne

Struktury danych

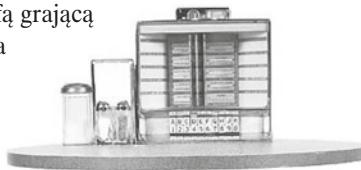


Rany... czyli w tych wszystkich przypadkach mogłem pozwolić Javie posortować całą zawartość w porządku alfabetycznym? Ten kurs jest do bani. Nigdy nie uczymy się niczego przydatnego.

Przechowywanie nie nastręcza w Javie żadnego problemu. Dysponujesz wszystkimi narzędziami związanymi z korzystaniem z kolekcji oraz sortowaniem danych i nie musisz pisać żadnych własnych algorytmów (no chyba że czytasz ten rozdział, siedząc na zajęciach kursu nauki o komputerach nr 101 — ponieważ w takim przypadku faktycznie BĘDZIESZ chciał pisać własny kod sortujący, natomiast reszta z nas ograniczy się do wywoływania metod udostępnianych przez klasy Javy). Biblioteka kolekcji Javy (*Java Collections Framework*) udostępnia struktury danych, które powinny zaspokoić praktycznie wszelkie Twoje potrzeby. Chciałbyś mieć listę, do której będziesz mógł bez problemów dodawać nowe elementy? Chcesz znaleźć coś na podstawie nazwy? Chcesz stworzyć listę, która będzie automatycznie eliminować powtarzające się elementy? Chcesz posortować listę współpracowników według liczby określającej, ile razy znienacka uderzyli Cię w plecy? Albo posortować listę ulubionych zwierząt na podstawie ilości sztuczek, jakie potrafią robić? To wszystko tu znajdziesz...

Śledzenie popularności piosenek w szafie grającej

Życzymy powodzenia w nowej pracy — zarządzaniu zautomatyzowaną szafą grającą w lokalu Obiady Leona. Sama szafa grająca ustawiona w tym lokalu nie ma wbudowanej Javy, jednak za każdym razem, gdy ktoś wybiera piosenkę, informacje o niej są dopisywane do zwyczajnego pliku tekstowego.



Twoim zadaniem jest wykorzystanie tych danych do określania popularności piosenek, generowania raportów i zarządzania listami odtwarzania. Nie musisz napisać całej aplikacji — w zadanie są zaangażowani także inni programiści-kelnerzy, jednak to właśnie Ty odpowiadasz za zarządzanie informacjami przechowywanymi w aplikacji napisanej w Javie i za ich sortowanie. A ponieważ Leon ma coś przeciwko bazom danych, wszystkie informacje są umieszczane w kolekcjach przechowywanych bezpośrednio w pamięci komputera. Jedyne, czym dysponujesz, to plik tekstowy, do którego szafa grająca wciąż dopisuje nowe piosenki. Twoje zadanie polega na „wyciągnięciu” informacji z tego pliku.

Już wiesz, jak należy odczytywać i analizować zawartość plików tekstowych, potrafisz już także przechowywać dane w kolekcji `ArrayList`.

ListaPiosenek.txt

```
Autobiografia/Perfect  
Zamki na piasku/Lady Pank  
Jolka, Jolka pamietasz/Budka Suflera  
Piosenka ksiezycowa/VARIUS MANX  
All My Love/LED ZEPPELIN  
Mniej niz zero/Lady Pank  
All You Need Is Love/The Beatles
```

Oto plik, który generuje szafa grająca.
Twój kod musi odczytać ten plik,
a następnie przetworzyć zapisane
w nim informacje o piosenkach.

Problem nr 1.

Posortowanie piosenek w kolejności alfabetycznej

W pliku zapisana jest lista piosenek, przy czym każdy wiersz zawiera informacje o jednej piosence — tytuł utworu oraz nazwę zespołu, oddzielone od siebie znakiem ukośnika. Przetworzenie wierszy tekstu o takiej postaci i zapisanie piosenek w kolekcji `ArrayList` powinno być łatwym zadaniem.

Twojego szefa interesują wyłącznie nazwy piosenek, więc na razie możesz ograniczyć się do utworzenia listy utworów.

Niestety, jak widać, lista nie jest zapisana w kolejności alfabetycznej. Co można z tym zrobić?

Jak wiesz, elementy w kolekcji `ArrayList` są przechowywane w takiej kolejności, w jakiej były dodawane. A zatem samo dodawanie ich do kolekcji nie zadba o to, by były posortowane alfabetycznie, chyba że... a może w klasie `ArrayList` jest jakaś metoda `sort()`¹?

¹ ang. sort oznacza „sortować”

Oto czym dysponujesz na razie, bez użycia sortowania:

```

import java.util.*;
import java.io.*;

public class SzafaGrajaca1 {

    ArrayList<String> listaPiosenek = new ArrayList<String>();

    public static void main(String[] args) {
        new SzafaGrajaca1().doDziela();
    }

    public void doDziela() { ←
        pobierzPiosenki();
        System.out.println(listaPiosenek);
    }

    void pobierzPiosenki() {
        try {
            File plik = new File("ListaPiosenek.txt");
            BufferedReader reader = new BufferedReader(new FileReader(plik));
            String wiersz = null;
            while ((wiersz = reader.readLine()) != null) {
                dodajPiosenke(wiersz);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void dodajPiosenke(String wierszDoAnalizy) {
        String[] elementy = wierszDoAnalizy.split("/");
        listaPiosenek.add(elementy[0]);
    }
}

```

Tytuły piosenek będziemy przechowywać w kolekcji `ArrayList` zawierającejłańcuchy znaków.

Ta metoda rozpoczyna od wczytywania pliku, a następnie wyświetla zawartość kolekcji „`listaPiosenek`”.

Tu nie ma nic szczególnego... po prostu odczytujemy zawartość pliku i dla każdego wiersza tekstu wywołujemy metodę `dodajPiosenke()`.

Metoda `dodajPiosenke()` działa tak samo jak tworzenie kart kwiżowych, które przedstawiliśmy w rozdziale poświęconym operacjom wejścia-wyjścia — dzielimy wiersz (zawierający zarówno tytuł piosenki, jak i nazwę zespołu) przy użyciu metody `split()`.

Interesuje nas wyłącznie tytuł piosenki, a zatem do „`listyPiosenek`” (`ArrayList`) dodajemy wyłącznie pierwszy element.

```

T:\>java SzafaGrajaca1
[Autobiografia, Zamki na piasku, Jolka, Jolka pamietasz, Piosenka ksiezycowa, All
My Love, Mniej niz zero, All You Need Is Love]
T:\>_

```

Jak widać, piosenki na liście są umieszczone w takiej samej kolejności, w jakiej były dodawane do kolekcji `ArrayList` (czyli takiej samej, w jakiej były zapisane w pliku tekstowym). Bez wątpienia NIE jest to kolejność alfabetyczna!

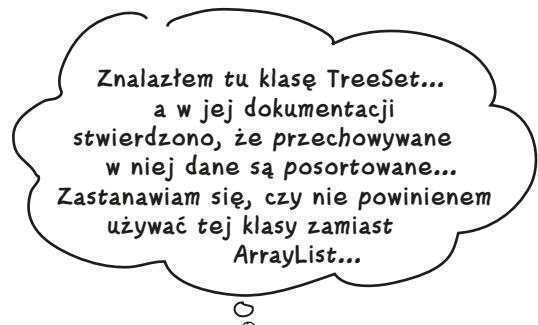
Ale klasa ArrayList NIE udostępnia metody sort()!

Kiedy popatrzymy na dokumentację klasy ArrayList, wydaje się, że nie ma w niej żadnej metody, która mogłaby posłużyć do posortowania zawartości kolekcji.

Przeanalizowanie klas bazowych w hierarchii dziedziczenia także niewiele nam w tym względzie pomoże — wyraźnie widać, że *na kolekcji ArrayList nie można wywołać żadnej metody sortującej*.

The screenshot shows a browser window displaying the Java 2 Platform SE 5.0 API documentation for the `ArrayList` class. The left sidebar lists various Java packages and classes. The main content area shows the `ArrayList` class definition with its methods listed in a table. A tooltip box is overlaid on the page, containing the text: "Klasa ArrayList ma wiele metod, jednak nie wydaje się, żeby którakolwiek z nich mogła służyć do sortowania...".

	Appends the specified element to the end of this list.
<code>void add(int index, E element)</code>	Inserts the specified element at the specified position in this list.
<code>boolean addAll(Collection<? extends E> c)</code>	Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specified Collection's Iterator.
<code>boolean addAll(int index, Collection<? extends E> c)</code>	Inserts all of the elements in the specified Collection into this list, starting at the specified position.
<code>void clear()</code>	Removes all of the elements from this list.
<code>Object clone()</code>	Returns a shallow copy of this <code>ArrayList</code> instance.
<code>boolean contains(Object elem)</code>	Returns true if this list contains the specified element.
<code>void ensureCapacity(int minCapacity)</code>	Increases the capacity of this <code>ArrayList</code> instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
<code>E get(int index)</code>	Returns the element at the specified position in this list.
<code>int indexOf(Object elem)</code>	Searches for the first occurrence of the given argument. Returns the index of the first occurrence of the specified element.
<code>boolean isEmpty()</code>	Tests if this list has no elements.
<code>int lastIndexOf(Object elem)</code>	Returns the index of the last occurrence of the specified element.
<code>E remove(int index)</code>	Removes the element at the specified position in this list.
<code>boolean remove(Object o)</code>	Removes a single instance of the specified element from this list, if it is present.
<code>protected void removeRange(int fromIndex, int toIndex)</code>	Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
<code>E set(int index, E element)</code>	Replaces the element at the specified position in this list with the specified element.
<code>int size()</code>	Returns the number of elements in this list.
<code>Object[] toArray()</code>	Returns an array containing all of the elements in this list in the correct order.
<code><T> T[] toArray(T[] a)</code>	



ArrayList nie jest jedyną dostępną kolekcją

Choć **ArrayList** jest kolekcją, której zapewne będziesz najczęściej używał, to jednak są także inne kolekcje nadające się do specjalnych zastosowań. Poniżej przedstawiliśmy kilka najważniejszych spośród nich:

➤ **TreeSet**

Sortuje elementy i zapobiega umieszczaniu w kolekcji elementów powtarzających się.

Nie musisz starać się zapamiętać tych innych kolekcji już teraz — zajmiemy się nimi nieco później.

➤ **HashMap**

Pozwala zapisywać pary elementów: nazwa-wartość i odwoływać się do nich.

➤ **LinkedList**

Zaprojektowana, by zapewnić lepszą efektywność dodawania i usuwania elementów ze środka kolekcji. (W praktyce w większości przypadków okazuje się, że kolekcja **ArrayList** i tak jest właśnie tym, czego będziesz chciał używać).

➤ **HashSet**

Zapobiega umieszczaniu w kolekcji powtarzających się elementów, poza tym, jeśli dysponujemy elementem, ta klasa pozwala na jego szybkie odszukanie w kolekcji.

➤ **LinkedHashSet**

Podobna do kolekcji **HashMap**, lecz potrafi zapamiętać kolejność, w jakiej elementy (pary nazwa-wartość) były dodawane; można ją także skonfigurować w taki sposób, by kolekcja pamiętała kolejność, w jakiej odwoływano się do jej elementów.

Mogłybyś użyć klasy TreeSet...

Lecz możesz także użyć metody Collections.sort()

Jeśli wszystkie łańcuchy znaków (tytuły piosenek) zapiszesz w kolekcji TreeSet, a nie ArrayList, to automatycznie znajdą się na odpowiednich miejscach, czyli zostaną posortowane alfabetycznie. Zawsze gdy zapragniesz wydrukować listę, piosenki pojawią się w kolejności alfabetycznej.

To doskonałe rozwiązanie, jeśli potrzebujesz *zbioru* (zajmiemy się nimi już niedługo), bądź też jeśli wiesz, że lista *zawsze* musi być posortowana alfabetycznie.

Jednak z drugiej strony, jeśli lista nie musi być zawsze posortowana, to korzystanie z klasy TreeSet może być zbyt kosztowne — *za każdym razem, gdy dodajesz element do kolekcji TreeSet, musi ona poświęcić trochę czasu, by określić, w jakim miejscu powinien się znaleźć nowy element*. W przypadku stosowania klasy ArrayList operacja dodawania nowego elementu do kolekcji może być wyjątkowo szybka, gdyż jest on po prostu umieszczany na końcu listy.

Nie istniejąca grupa pytania

P: Jednak **ISTNIEJE** możliwość dodania elementu w określonym miejscu kolekcji ArrayList, a nie tylko na jej końcu — dostępna jest przeciążona metoda add(), która oprócz dodawanego elementu pobiera także liczbę całkowitą. Czy w takim przypadku dodawanie elementu w wybranym miejscu jest wolniejsze od dodawania go na końcu listy?

O: Tak, umieszczanie elementów w kolekcji ArrayList w innym miejscu niż na końcu jest wolniejsze. Oznacza to, że wykonanie przeciążonej metody add(indeks, element) trwa dłużej niż wykonanie metody add(element) — która to umieszcza nowy element na końcu kolekcji. Jednak w większości przypadków, korzystając z kolekcji ArrayList, nie będziesz musiał dodawać elementów w określonych miejscach.

P: Jak widzę, dostępna jest także klasa LinkedList, a zatem, czy ona nie nadawałaby się lepiej do dodawania elementów do środka kolekcji? Tak przynajmniej pamiętam ze studiów, z zajęć poświęconych strukturom danych.

O: Owszem, masz rację. Klasa LinkedList może zapewniać lepszą efektywność dodawania i usuwania elementów ze środka kolekcji, jednak dla przeważającej większości aplikacji różnica szybkości dodawania elementów do środka kolekcji LinkedList i ArrayList nie jest na tyle duża, by warto ją uwzględnić. Zaczyna ona odgrywać rolę, gdy operujemy na naprawdę ogromnej ilości elementów. Kolejką LinkedList zajmiemy się już niebawem.

java.util.Collections

```
public static void copy(List destination, List source)
public static List emptyList()
public static void fill(List listToFill, Object objToFillItWith)
public static int frequency(Collection c, Object o)
public static void reverse(List list)
public static void rotate(List list, int distance)
public static void shuffle(List list)
public static void sort(List list)
    !! List list, Object oldVal, Object newVal)
public static boolean
// wiele innych metod
```

Hmm... okazuje się, że w klasie Collections JEST dostępna metoda sort(). Wymaga ona podania argumentu typu List, a ponieważ klasa ArrayList implementuje interfejs List, zatem ArrayList JEST typu List. Dzięki polimorfizmowi możemy przekazać ArrayList w wywołaniu metody oczekującej argumentu typu List.

Uwaga: To NIE jest lista prawdziwych metod klasy Collections — uprościliśmy ją, usuwając informacje o typie ogólnym (dojdziemy do nich już niedługo).

Dodanie wywołania Collections.sort() do kodu aplikacji SzafaGrajaca

```

import java.util.*;
import java.io.*;

public class SzafaGrajaca2 {

    ArrayList<String> listaPiosenek = new ArrayList<String>();

    public static void main(String[] args) {
        new SzafaGrajaca2().doDziela();
    }

    public void doDziela() {
        pobierzPiosenki();
        System.out.println(listaPiosenek);
        Collections.sort(listaPiosenek);
        System.out.println(listaPiosenek); ←
    } ←
}

void pobierzPiosenki() {
    try {
        File plik = new File("ListaPiosenek.txt");
        BufferedReader reader = new BufferedReader(new FileReader(plik));
        String wiersz = null;
        while ((wiersz= reader.readLine()) != null) {
            dodajPiosenke(wiersz);
        }
    } catch(Exception ex) {
        ex.printStackTrace();
    }
}

void dodajPiosenke(String wierszDoAnalizy) {
    String[] elementy = wierszDoAnalizy.split("/");
    listaPiosenek.add(elementy[0]);
}
}

```

Metoda `Collections.sort()` sortuje listę łańcuchów znaków w kolejności alfabetycznej.

Wywołujemy statyczną metodę `Collections.sort()`, a następnie ponownie wyświetlamy listę. Za drugim razem będzie ona posortowana alfabetycznie!

```

T:\>java SzafaGrajaca1
[Autobiografia, Zamki na piasku, Jolka, Jolka pamietasz, Piosenka ksiezycowa, Al
l My Love, Mniej niz zero, All You Need Is Love]
[All My Love, All You Need Is Love, Autobiografia, Jolka, Jolka pamietasz, Mniej
niz zero, Piosenka ksiezycowa, Zamki na piasku]

T:\>_

```

Przed wywołaniem metody `sort()`. ←
Po wywołaniu metody `sort()`. ←

Sortowanie własnych obiektów

Teraz jednak potrzebujesz obiektów Piosenka, a nie zwykłych łańcuchów znaków

A teraz Twój szef chce, by na liście były przechowywane obiekty klasy Piosenka, a nie same łańcuchy znaków. W ten sposób można by zgromadzić więcej informacji o poszczególnych piosenkach. Nowa szafa grająca publikuje więcej informacji o piosenkach, a zatem teraz w każdym wierszu pliku tekstowego będą zapisane cztery, a nie dwa elementy.

Klasa Piosenka jest naprawdę prosta, ma tylko jedną interesującą cechę — przesłoniętą metodę `toString()`. Pamiętasz, że metoda `toString()` jest zdefiniowana w klasie `Object`, zatem dziedziczy ją każda klasa w Javie. A ponieważ metoda ta jest wywoływana dla każdego obiektu wyświetlanego przy użyciu metod `print()` (także `System.out.println()`), powinieneś przesłonić ją, by wyświetlać informacje nieco bardziej zrozumiałe niż unikalny kod identyfikujący obiekt. W przypadku wyświetlania listy metoda `toString()` zostanie wywołana dla każdego obiektu przechowywanego w kolekcji.

```
class Piosenka {  
  
    String tytul;  
    String artysta;  
    String ocena;  
    String bpm;  
  
    Piosenka(String t, String a, String o, String b) {  
        tytul = t;  
        artysta = a;  
        ocena = o;  
        bpm = b;  
    }  
  
    public String getTytul() {  
        return tytul;  
    }  
  
    public String getArtysta() {  
        return artysta;  
    }  
  
    public String getOcena() {  
        return ocena;  
    }  
    public String.getBpm() {  
        return bpm;  
    }  
  
    public String toString() { ←  
        return tytul;  
    }  
}
```

Cztery składowe służące do przechowywania czterech atrybutów piosenki odczytanych z pliku.

Wartości wszystkich zmiennych są określane w konstruktorze podczas tworzenia obiektu Piosenka.

Metody pobierające wartości poszczególnych atrybutów.

Przestaniemy metodę `toString()`, ponieważ gdy będziemy wyświetlać obiekt przy użyciu wywołania `System.out.println(piosenka)`, chcemy, by pojawiła się jej nazwa. W efekcie wywołania `System.out.println(listaPiosenek)` dla każdego elementu listy zostanie wywołana metoda `toString()`.

ListaPiosenek.txt

```
Autobiografia/Perfect/7/80  
Zamki na piasku/Lady Pank/5/120  
Jolka, Jolka pamietasz/Budka Suflera/9/100  
Piosenka ksiezycowa/VARIUS MANX/7/90  
All My Love/LED ZEPPELIN/9/120  
Mniej niz zero/Lady Pank/7/100  
All You Need Is Love/The Beatles/10/90
```

Nowy plik piosenek zawiera dla każdej z nich nie dwa, lecz cztery atrybuty. Chcemy, by WSZYSTKIE z nich znalazły się na naszej liście, dlatego też będziemy musieli utworzyć klasę Piosenka ze składowymi dla wszystkich czterech atrybutów.

Zastąpieniełańcuchów znaków obiektami Piosenka

W Twoim kodzie trzeba wprowadzić bardzo nieznaczne poprawki — kod obsługujący odczyt i analizę pliku tekstowego jest taki sam (bazuje na metodzie `String.split()`); różnica polega na tym, że dla każdego wiersza uzyskujemy *cztery* elementy i wszystkie z nich zostaną użyte do utworzenia nowego obiektu `Piosenka`. No i oczywiście kolekcja `ArrayList` będzie teraz przechowywać obiekty `Piosenka`, a nie `String`.

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

public class SzafaGrajaca3 {
    ArrayList<Piosenka> listaPiosenek = new ArrayList<Piosenka>();

    public static void main(String[] args) {
        new SzafaGrajaca3().doDziela();
    }

    public void doDziela() {
        pobierzPiosenki();
        Collections.sort(listaPiosenek);
        System.out.println(listaPiosenek);
    }

    void pobierzPiosenki() {
        try {
            File plik = new File("ListaPiosenek.txt");
            BufferedReader reader = new BufferedReader(new FileReader(plik));
            String wiersz = null;
            while ((wiersz= reader.readLine()) != null) {
                dodajPiosenke(wiersz);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void dodajPiosenke(String wierszDoAnalizy) {
        String[] elementy = wierszDoAnalizy.split("/");
        Piosenka nastepnaPiosenka = new Piosenka(elementy[0], elementy[1], elementy[2], elementy[3]);
        listaPiosenek.add(nastepnaPiosenka);
    }
}

Tworzymy kolekcję ArrayList<Piosenka>, zawierającą obiekty Piosenka, a nie łańcuchy znaków.
```

Tworzymy nowy obiekt `Piosenka`, używając przy tym wszystkich czterech elementów (czyli czterech fragmentów podzielonego wiersza tekstu pobranego z pliku tekstowego); następnie dodajemy piosenkę do kolekcji.

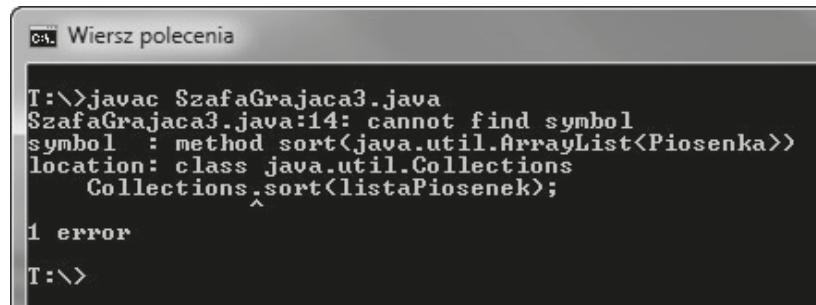
To się nie chce skompilować!

Coś jest źle... w dokumentacji klasy Collections wyraźnie widać, że jest dostępna metoda sort() pobierająca argument typu List.

Klasa ArrayList jest typu List, gdyż implementuje ona interfejs List... a zatem... to powinno działać.

Ale nie działa!

Kompilator twierdzi, że nie może znaleźć metody sort() pobierającej argument typu ArrayList<Piosenka>. Może kompilator nie lubi kolekcji obiektów Piosenka? Wcześniej nie miał nic przeciwko kolekcji ArrayList<String>... a zatem cóż jest tak istotnego, co odróżnia klasę String od klasy Piosenka? Na czym polega różnica, która sprawia, że kompilator odmawia skompilowania nowej wersji kodu?



The screenshot shows a terminal window titled "Wiersz polecenia". The command entered is "javac SzafaGrajaca3.java". The output shows a single error: "SzafaGrajaca3.java:14: cannot find symbol symbol : method sort<java.util.ArrayList<Piosenka>> location: class java.util.Collections Collections.sort(listaPiosenek); ^ 1 error". The prompt "T:>" is visible at the bottom.

Zapewne zadałeś już sobie pytanie: „na podstawie *czego* będziemy sortować piosenki?”. Niby skąd metoda sort() może w ogóle wiedzieć, co sprawia, że jedna piosenka może być większa lub mniejsza od innej? Bez wątpienia, jeśli chcesz, by to *tytuł* stanowił wartość determinującą kolejność sortowania piosenek, to będziesz potrzebował jakiegoś sposobu, by poinformować metodę sortującą, że ma operować na tytułe, a nie, powiedzmy, na ocenie piosenki.

Wszystkimi tymi zagadnieniami zajmiemy się już za kilka stron, najpierw jednak dowiedzmy się, dlaczego kompilator nie pozwolił nam przekazać kolekcji ArrayList z obiektami Piosenka w wywołaniu metody sort().



Deklaracja metody sort()

The screenshot shows a Mozilla Firefox browser window displaying the Java 2 Platform SE 5.0 API documentation. The URL in the address bar is http://download.oracle.com/docs/cd/E17476_01/javase/1.5.0/docs/api/. The page title is "Java 2 Platform SE 5.0 - Mozilla Firefox". On the left, there is a navigation sidebar with categories like "java.util", "Observer", "Queue", "RandomAccess", "Set", "SortedMap", "SortedSet", "Classes", "AbstractCollection", "AbstractList", and "AbstractMap". The main content area is titled "Method Detail" and shows the declaration of the "sort" method:

```
public static <T extends Comparable<? super T> void sort(List<T> list)
```

The documentation text states: "Sorts the specified list into ascending order, according to the *natural ordering* of its elements. All elements in the list must implement the Comparable interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list). This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort."

W dokumentacji Javy widać (po wyświetleniu strony dotyczącej klasy `java.util.Collections` i przewinięciu jej tak, by były widoczne informacje o metodzie `sort()`), że metoda `sort()` jest zadeklarowana... *dziwnie*. Albo przynajmniej zupełnie inaczej niż wszystko, z czym się do tej pory spotkaliśmy.

Wynika to z faktu, że metoda `sort()` (oraz wszystkie inne elementy biblioteki kolekcji) w bardzo dużym stopniu wykorzystuje *typy ogólne* (ang. *generics*). Za każdym razem, gdy w kodach źródłowych lub dokumentacji Javy zobaczysz coś zapisanego w nawiasach kątowych, mogą to być właśnie typy ogólne — nowa możliwość języka dodana w wersji 5.0 Javy. Wygląda zatem na to, że zanim będziemy w stanie zrozumieć, dlaczego mogliśmy sortować kolekcję `ArrayList` zawierającą łańcuchy znaków, lecz nie mogliśmy sortować tej samej kolekcji z obiektami `Piosenka`, będziemy musieli nauczyć się, jak interpretować dokumentację.

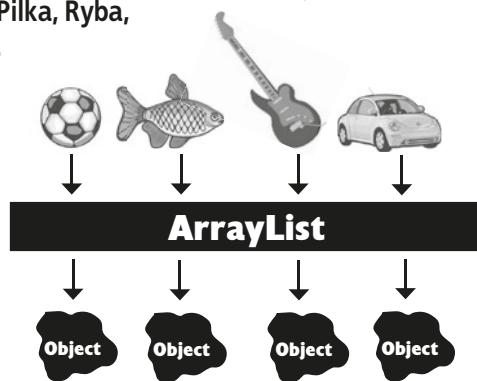
Typy ogólne oznaczają większe bezpieczeństwo typów

Napiszemy to tu bez żadnych niedomówień — praktycznie cały kod wykorzystujący typy ogólne, który będziesz pisał, będzie związany z wykorzystaniem kolekcji. Choć typy ogólne mogą być także stosowane na inne sposoby, to jednak ich podstawowym przeznaczeniem jest udostępnienie możliwości tworzenia kolekcji zapewniających wysoki poziom bezpieczeństwa typów. Innymi słowy, chodzi o kod napisany w taki sposób, by kompilator nie pozwolił Ci umieścić obiektu Pies tam, gdzie jest oczekiwany obiekt Kaczka.

Przed wprowadzeniem typów ogólnych (czyli przed wersją 5.0 Javy) kompilator nie zwracał wielkiej uwagi na to, co umieszczasz w kolekcjach, gdyż wszystkie implementacje kolekcji operowały na obiektach typu Object. W kolekcji *ArrayList* można było umieścić praktycznie wszystko; zupełnie tak, jak gdyby wszystkie kolekcje *ArrayList* były deklarowane jako *ArrayList<Object>*.

BEZ typów ogólnych

Obiekty są zapisywane jako odwołania typów Pilka, Ryba, Gitara i Samochod.



A odczytywano je jako odwołania typu Object.

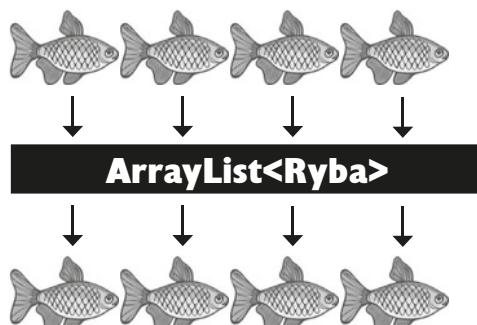
Przed wprowadzeniem typów ogólnych nie można było zadeklarować typu zawartości kolekcji *ArrayList*, dlatego też jej metoda *add()* operowała na obiektach typu *Object*.

Dzięki typom ogólnym możesz tworzyć kolekcje zapewniające większe bezpieczeństwo typów, w których więcej błędów jest wyłapywanych w czasie komplikacji, a nie działania programu.

Bez wykorzystania typów ogólnych kompilator radośnie pozwoli Ci zapisać obiekt Dynia w kolekcji, która miała przechowywać jedynie obiekty Auto.

Z wykorzystaniem typów ogólnych

Obiekty są zapisywane jako odwołania do jednej klasy — Ryba



I odczytuje się je jako odwołania typu Ryba.

Aktualnie, dzięki zastosowaniu typów ogólnych, w kolekcji zadeklarowanej jako *ArrayList<Ryba>* umieszczać można wyłącznie obiekty klasy *Ryba*, dzięki czemu obiekty są pobierane z kolekcji jako odwołania typu *Ryba*. Nie musisz się zatem przejmować, że ktoś zapisze w takiej kolekcji obiekt *Volkswagen*, ani że obiektu, który odczytasz z kolekcji, nie będzie można prawidłowo rzutować do typu *Ryba*.

Poznajemy typy ogólne

Spośród kilkunastu zagadnień dotyczących typów ogólnych, które mógłbyś poznać, tak naprawdę tylko trzy mają duże znaczenie dla większości programistów.

1 Tworzenie obiektów klas uogólnionych (takich jak `ArrayList`)

Tworząc kolekcję `ArrayList`, musisz określić typ obiektów, które będą mogły być na niej zapisywane; analogicznie jak to robimy, tworząc zwyczajne tablice.

```
new ArrayList<Piosenka>()
```

2 Deklarowanie i przypisywanie zmiennych typów ogólnych

Jak działa polimorfizm w przypadku typów ogólnych? Czy jeśli dysponujesz zmienną referencyjną typu `ArrayList<Zwierze>`, to możesz zapisać w niej odwołanie do kolekcji `ArrayList<Pies>`? A co ze zmienną typu `List<Zwierze>`? Czy można w niej zapisać odwołanie typu `ArrayList<Zwierze>`? Dowiesz się już niebawem...

```
List<Piosenka> listaPiosenek =  
    new ArrayList<Piosenka>()
```

3 Deklarowanie (i wywoływanie) metod mających argumenty typu ogólnego

Jeśli dysponujesz metodą, której parametr jest, na przykład, kolekcją `ArrayList` zawierającą obiekty `Zwierze`, to co to tak naprawdę oznacza? Czy możesz w jej wywołaniu przekazać kolekcję `ArrayList` obiektów `Pies`? Przyjrzymy się pewnym subtelnym i złożonym zagadnieniom związанныm z polimorfizmem i typami ogólnymi, które sprawiają, że pisanie metody z argumentami typów ogólnych znaczco różni się od tworzenia metod oczekujących argumentów będących zwyczajnymi tablicami.

```
void fu(List<Piosenka> lista)
```

```
x.fu(listaPiosenek)
```

(W rzeczywistości jest to to samo zagadnienie, co w punkcie nr 2, lecz w ten sposób pokazujemy, jak wielkie, według nas, jest jego znaczenie).

^{Nie, istnieja} głupie pytania

P: A czy nie powiniem się także nauczyć, jak tworzyć swoje **WŁASNE typy ogólne?**
Co mam zrobić, gdybym chciał napisać klasę, która pozwalałaby programistom określić, jakie będą typy używanych w niej danych?

O: Zapewne nieczęsto będziesz miał takie potrzeby. Zastanów się — projektanci biblioteki Javy stworzyli całą grupę klas kolekcji obejmujących niemal wszystkie struktury danych, jakich możesz potrzebować; a w praktyce jedynym typem klas, które muszą być uogólnione, są właśnie klasy kolekcji. Innymi słowy — klasy zaprojektowane po to, by przechowywały elementy innych typów. Jednocześnie chcemy, by programiści używający takich kolekcji podczas ich deklarowania i tworzenia określali, jaki będzie typ ich zawartości.

Owszem, istnieje takie prawdopodobieństwo, że będziesz chciał napisać własną klasę uogólnioną, jednak jest to sytuacja wyjątkowa i z tego względu nie opiszymy jej w tej książce. (Niemniej jednak i tak domyślisz się, jak należy to robić, na podstawie innych, zamieszczonych w tekście informacji).

Używanie KLAS uogólnionych

Klasa `ArrayList` jest najczęściej używanym przez nas typem uogólnianym, dlatego też zaczniemy od przeglądu jej dokumentacji. Istnieją dwie kluczowe informacje o klasie uogólnionej, na które warto zwrócić uwagę:

1. Deklaracja klasy.
2. Deklaracje metod służących do dodawania nowych elementów.

Zrozumieć dokumentację klasy `ArrayList`

(Albo inaczej — jakie jest prawdziwe znaczenie litery „E”?)

Wyobraź sobie, że „E” oznacza „typ elementu, który chcesz przechowywać w kolekcji i który ma ona zwracać” (E oznacza Element).

```
public class ArrayList<E> extends AbstractList<E> implements List<E> ... {  
    public boolean add(E o) ...  
    // dalszy kod klasy  
}
```

„E” jest symbolem zastępującym PRAWDZIWY typ, który podasz, deklarując i tworząc obiekt `ArrayList`.

ArrayList jest klasą potomną `AbstractList`, a za tem dowolny typ, jakiego użyjesz w kolekcji `ArrayList`, zostanie także automatycznie zastosowany jako typ dla klasy `AbstractList`.

To jest najważniejsze! Niezależnie od tego, czym jest typ „E”, właśnie on będzie typem elementów, jakie będzie można dodawać do kolekcji.

Typ (wartość `<E>`) staje się także typem dla interfejsu.

„E” reprezentuje typ, który zostanie użyty podczas tworzenia obiektu `ArrayList`. Kiedy zobaczysz tę literę w dokumentacji kolekcji, możesz ją w myślach zastąpić tym `<typem>`, którego użyjesz podczas tworzenia kolekcji `ArrayList`.

A zatem, jeśli zastosujesz wyrażenie `ArrayList<Piosenka>`, we wszystkich metodach i zmiennych, w jakich występuje litera „E”, zostanie ona zastąpiona przez „Piosenka”.

Stosowanie parametrów typu w kolekcji ArrayList

PONIŻSZY kod:

```
ArrayList<String> lista = new ArrayList<String>
{
    public class ArrayList<E> extends AbstractList<E> ... {
        public boolean add(E o)
        // dalszy kod klasy
    }
}
```

Oznacza, że kolekcja ArrayList:

The diagram consists of three curved arrows. One arrow points from the first 'E' in 'ArrayList<String>' to the 'E' in 'ArrayList<E>'. Another arrow points from the second 'E' in 'ArrayList<String>' to the 'E' in 'AbstractList<E>'. A third arrow points from the 'E' in 'add(E o)' to the 'E' in 'public boolean add(E o)'.

Zostanie potraktowana przez kompilator jako:

```
public class ArrayList<String> extends AbstractList<String> ... {

    public boolean add(String o)
    // dalszy kod klasy
}
```

Innymi słowy, litera „E” zostanie zastąpiona przez *prawdziwy typ* (nazywany także *parametrem typu*), którego użyjesz podczas tworzenia obiektu ArrayList. I właśnie dlatego metoda add() tej klasy pozwoli Ci dodać do kolekcji wyłącznie obiekty typów referencyjnych zgodnych z „E”. A zatem, jeśli utworzysz kolekcję ArrayList<String>, to metoda add() stanie się nagle metodą add(String o). Jeśli natomiast utworzysz kolekcję obiektów Pies, metoda add() zmieni się w metodę add(Pies o).

Nie istnieją
głupie pytania

P: Czy „E” to jedyny typ, jaki mogę stosować? Pytam, bo w dokumentacji metody sort() pojawiła się litera „T”...

G: Możesz użyć wszystkiego, co z punktu widzenia języka Java jest prawidłowym identyfikatorem. A zatem wszystko, czego mógłbyś użyć jako nazwy metody lub zmiennej, możesz także zastosować jako parametr typu. Zwyczajowo stosowana jest jednak tylko jedna litera (a zatem także i Ty powinieneś się do tego dostosować), co więcej, powinna to być litera „T”, chyba że tworzona klasa jest kolekcją — w tym przypadku konwencja nakazuje użycie litery „E” oznaczającej „typ Elementów kolekcji”.

Stosowanie METOD uogólnionych

Określenie *klasa* uogólniona oznacza, że deklaracja klasy zawiera parametr typu.
Z kolei określenie *metoda* uogólniona (ang. *generic method*) oznacza, że w deklaracji metody, w jej sygnaturze, zastosowano parametr typu.

Parametrów typu można używać w deklaracjach metod na kilka różnych sposobów:

1 Stosowanie parametru typu zdefiniowanego w deklaracji klasy

```
public class ArrayList<E> extends AbstractList<E> ... {  
    public boolean add(E o)
```

Możesz zastosować tu literę „E” wyłącznie dlatego, że została użyta wcześniej jako fragment deklaracji klasy.

W przypadku gdy deklarujesz parametr typu dla klasy, możesz go potem używać wszędzie tam, gdzie normalnie używałbyś *rzeczywistych* nazw klas lub interfejsów. Typ zadeklarowanego argumentu metody zostanie zastąpiony typem, którego użyjesz podczas tworzenia obiektu klasy uogólnionej.

2 Stosowanie parametru typu, który NIE został zdefiniowany w deklaracji klasy

```
public <T extends Zwierze> void wezNaRece(ArrayList<T> lista)
```

W tym miejscu możemy użyć typu „T”, gdyż wcześniej zadeklarowaliśmy „T” w deklaracji metody.

Jeśli sama klasa nie używa parametru typu, to i tak możesz określić taki dla konkretnej metody — należy go zadeklarować w bardzo niezwykłym (choć dostępnym) miejscu — *przed typem wartości wynikowej*.

Ta deklaracja metody stwierdza, że T może być dowolnym typem potomnym typu Zwierze.



Chwila... tu musi być jakiś błąd...
 Jeśli możesz przekazać listę obiektów
 Zwierze, dlaczego nie można tego
 WYRAZIĆ precyzyjnie? Co jest złego
 w zadeklarowaniu metody w następujący
 sposób: `wezNaRece(ArrayList<Zwierze> lista)`?

W tym miejscu wszystko się komplikuje...

Ta deklaracja:

```
public <T extends Zwierze> void wezNaRece(ArrayList<T> lista)
```

NIE oznacza tego samego, co ta:

```
public void wezNaRece(ArrayList<Zwierze> lista)
```

Obie są prawidłowe, jednak oznaczają co innego!

Pierwsza z nich, ta zawierająca człon `<T extends Zwierze>`, oznacza, że w wywołaniu metody można przekazać dowolną kolekcję `ArrayList`, zadeklarowaną jako kolekcja obiektów `Zwierze` lub obiektów jej klasy potomnej (takiej jak `Pies` lub `Kot`). A zatem tę górną metodę mógłbyś wywołać, przekazując jako argument kolekcję `ArrayList<Pies>`, `ArrayList<Kot>` lub `ArrayList<Zwierze>`.

Z kolei druga deklaracja metody, ta, w której argument ma postać `ArrayList<Zwierze>`, oznacza, że *jedynym* dozwolonym argumentem jest kolekcja `ArrayList<Zwierze>`. Innymi słowy, w wywołaniu pierwszej metody możemy przekazać dowolną kolekcję zawierającą obiekty, które SĄ Zwierzęciem (czyli `Zwierze`, `Pies`, `Kot` i tak dalej), natomiast w drugiej można przekazać *wyłącznie* kolekcję zawierającą obiekty typu `Zwierze`... Nie `ArrayList<Pies>` czy `ArrayList<Kot>`, lecz tylko `ArrayList<Zwierze>`.

I faktycznie, można się zgodzić z twierdzeniem, że jest to sprzeczne z ideą polimorfizmu, jednak przyczyny takiego stanu rzeczy staną się oczywiste pod koniec rozdziału, gdy zajmiemy się tym zagadnieniem bardziej szczegółowo. Na razie pamiętaj, proszę, że zajmujemy się tym problemem wyłącznie dlatego, że wciąż staramy się dowiedzieć, jak mamy posortować kolekcję `Piosenek` — w tym celu musielibyśmy zatrzymać się na dokumentacji metody `sort()`, w której zobaczyliśmy tę dziwaczna deklarację wykorzystującą typy ogólne.

Jak na razie wystarczy, byś wiedział, że składnia pierwszej metody podanej na tej stronie jest prawidłowa i oznacza, że do metody można przekazać obiekt `ArrayList` zdefiniowany jako kolekcja obiektów `Zwierze` lub klas potomnych.

A teraz wróćmy do naszej metody `sort()`...

Sortowanie piosenek



To wciąż nie wyjaśnia, dlaczego metoda sort() działała na kolekcji obiektów String, a nie chciała działać w przypadku użycia kolekcji ArrayList z obiektami Piosenka.

Pamiętasz, w jakim miejscu skończyliśmy?

```
cmd Wiersz polecenia
T:\>javac SzafaGrajaca3.java
SzafaGrajaca3.java:14: cannot find symbol
symbol : method sort(java.util.ArrayList<Piosenka>)
location: class java.util.Collections
    Collections.sort(listaPiosenek);
                ^
1 error
T:\>
```

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

public class SzafaGrajaca3 {

    ArrayList<Piosenka> listaPiosenek = new ArrayList<Piosenka>();

    public static void main(String[] args) {
        new SzafaGrajaca3().doDziela();
    }

    public void doDziela() {
        pobierzPiosenki();
        Collections.sort(listaPiosenek);
        System.out.println(listaPiosenek);
    }

    void pobierzPiosenki() {
        try {
            File plik = new File("ListaPiosenek.txt");
            BufferedReader reader = new BufferedReader(new FileReader(plik));
            String wiersz = null;
            while ((wiersz= reader.readLine()) != null) {
                dodajPiosenke(wiersz);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void dodajPiosenke(String wierszDoAnalizy) {
        String[] elementy = wierszDoAnalizy.split("/");
        Piosenka nastepnaPiosenka = new Piosenka(elementy[0], elementy[1], elementy[2], elementy[3]);
        listaPiosenek.add(nastepnaPiosenka);
    }
}
```

Właśnie w tym miejscu kompilator zaprotestował!
Nie miał żadnych obiektów, kiedy przekazywaliśmy
kolekcję ArrayList<String>, a kiedy przekazaliśmy
ArrayList<Piosenka> — momentalnie wyświetlił błąd.

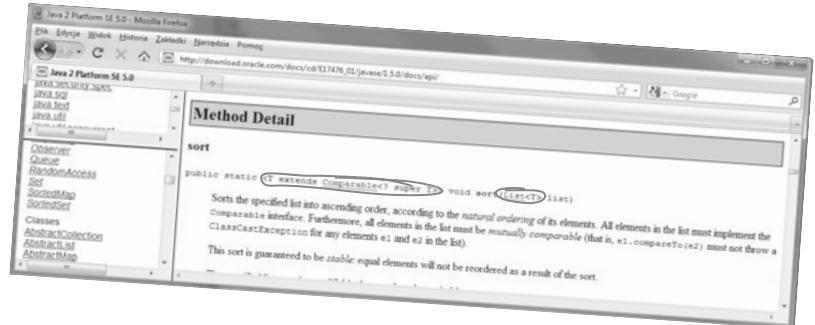
Ponownie odwiedzimy metodę sort()

No i jesteśmy z powrotem w tym samym miejscu, starając się przeczytać dokumentację metody sort() i zrozumieć, dlaczego mogliśmy sortować kolekcję łańcuchów znaków (String), a nie możemy posortować kolekcji obiektów Piosenka. I wygląda na to, że odpowiedź na to pytanie brzmi...

Do metody sort() można przekazać wyłącznie listę obiektów klasy Comparable lub potomnej.

Piosenka nie jest klasą potomną Comparable i dlatego nie można posortować listy piosenek.

Przynajmniej jeszcze nie teraz...



`public static <T extends Comparable<? super T>> void sort(List<T> list)`

To oznacza: „Niezależnie od tego, czym jest ‐T‐, musi ono być typu Comparable”.

(Tę część możesz na razie zignorować. Jeśli jednak nie możesz tego zrobić, to wiedz, że oznacza to: ‐parametr typu dla obiektu Comparable musi być typu T lub jego typu potomnego‐).

Możemy przekazać tylko listę (List, lub jej klasę potomną, taką jak ArrayList) utworzoną przy użyciu typu parametrycznego ‐roszszerzającego Comparable‐.

Hm... Sprawdzitam dokumentację klasy String – okazuje się, że ona nie rozszerza Comparable – ona go IMPLEMENTUJE. „Go” ponieważ Comparable to interfejs. A zatem stwierdzenie <T extends Comparable> nie ma sensu.



```
public final class String extends Object implements Serializable,
    Comparable<String>, CharSequence
```

W przypadku typów ogólnych „rozszerzać” oznacza „rozszerzać lub implementować”

Twórcy Javy musieli zapewnić nam jakąś możliwość nakładania ograniczeń na typy sparametryzowane, byśmy mogli pozwolić na użycie na przykład jedynie obiektów klas potomnych klasy `Zwierze`. Jednak potrzebna jest także możliwość nakładania ograniczeń, które będą zezwalały na stosowanie wyłącznie tych klas, które implementują podany interfejs. A zatem mamy sytuację, w której potrzebujemy jednej i tej samej konstrukcji składniowej, którą można by wykorzystać w obu przypadkach — zarówno dziedziczenia, jak i implementacji. Innymi słowy, która działa w obu sytuacjach — gdy chcemy coś *rozszerzać* lub coś *implementować*.

Ostatecznie zdecydowano się na... *rozszczarzanie*. Jednak w tym przypadku „rozszczarzanie” oznacza relację „jest” i można je stosować niezależnie od tego, czy typ umieszczony z jego prawej strony jest interfejsem, czy też klasą.

W przypadku typów ogólnych słowo kluczowe „extends” w rzeczywistości oznacza relację „jest” i odnosi się ZARÓWNO do klas, JAK I do interfejsów.

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Nie ma znaczenia, czy typ umieszczony z prawej strony jest klasą, czy interfejsem... w obu przypadkach mówimy o jego „rozszczarzaniu”.

Comparable jest interfejsem, a zatem tak NAPRAWDĘ chodzi nam o to, że „T musi być typem, który implementuje interfejs Comparable”.

P: Dlaczego twórcy języka po prostu nie wprowadzili nowego słowa kluczowego: „is” (ang. *jest*)?

O: Dodanie nowego słowa kluczowego do języka programowania jest naprawdę DUŻYM wyzwaniem, gdyż zawsze pojawia się ryzyko, że doprowadzi ono do pojawienia się błędów w już istniejącym kodzie. Zastanów się — być może używałeś zmiennej „is”. A ponieważ nie można używać słów kluczowych jako identyfikatorów, oznacza to, że w każdym kodzie, w którym takie słowo było używane, zanim stało się słowem kluczowym, pojawią się błędy. Dlatego też zawsze, gdy tylko nadarza się okazja do wykorzystania istniejących słów kluczowych, projektanci Javy starają się tak robić. I tak właśnie postąpili w przypadku słowa `extends`. Jednak nie zawsze istnieje taka możliwość.

Do Javy dodano niewiele (naprawdę bardzo niewiele) nowych słów kluczowych, takich jak **assert** (w Javie 1.4) czy też **enum** w Javie 1.5 (tym nowym słowem kluczowym zajmiemy się w dodatku

W przypadku typów ogólnych słowo kluczowe „extends” w rzeczywistości oznacza relację „jest” i odnosi się ZARÓWNO do klas, JAK I do interfejsów.

Nie istnieją grupie pytania

umieszczonym na końcu tej książki). Te zmiany naprawdę spowodowały problemy w niektórych, już istniejących, kodach. Jednak czasami istnieje możliwość skompilowania i uruchomienia programu przy użyciu nowszej wersji Javy w taki sposób, by zachowywała się ona jak jedna z wcześniejszych wersji języka. W tym celu należy z poziomu wiersza poleceń przekazać do kompilatora lub JVM specjalną flagę, która stwierdza: „No dobra, ja wiem, że jesteś wersją 1.4 Javy, ale proszę, poudawaj, że jesteś wersją 1.3... Wiesz, w kodzie mojego programu używam zmiennej o nazwie `assert`, którą zacząłem stosować jeszcze jak twoje kody bajtowe były w powijakach”.

(Aby przekonać się, czy ta flaga jest dostępna, wpisz w wierszu poleceń `javac` (jeśli chcesz sprawdzić kompilator) lub `java` (jeśli interesuje Cię JVM), bez żadnych dodatkowych argumentów. Na ekranie powinna zostać wyświetlona lista wszystkich dostępnych argumentów.Więcej informacji o tych flagach przedstawimy w rozdziale poświęconym publikowaniu i udostępnianiu programów napisanych w Javie.

W końcu wiemy, w czym tkwi problem...

Piosenka musi implementować interfejs Comparable

Kolekcję `ArrayList<Piosenka>` będziemy mogli przekazać w wywołaniu metody `sort()` wyłącznie w przypadku, jeśli w klasie `Piosenka` zostanie zaimplementowany interfejs `Comparable`. Właśnie w taki sposób została bowiem zadeklarowana metoda `sort()`. Krótkie sprawdzenie dokumentacji pozwala nam ustalić, że `Comparable` jest naprawdę prostym interfejsem, deklarującym tylko jedną metodę:

`java.lang.Comparable`

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

A oto zamieszczone w dokumentacji informacje dotyczące metody `compareTo()`:

Wartość wynikowa:

Liczba całkowita mniejsza od zera, zero lub liczba całkowita większa od zera zwracane odpowiednio w sytuacjach, gdy ten obiekt jest mniejszy, równy lub większy od obiektu przekazanego w wywołaniu.

Wygląda na to, że metoda `compareTo()` zostanie wywołana na rzecz jednego obiektu `Piosenka`, a argumentem jej wywołania będzie drugi obiekt `Piosenka`. Obiekt `Piosenka` użyty do wywołania metody `compareTo()` musi określić, czy przekazany do metody obiekt powinien znaleźć się na liście przed nim, w tym samym miejscu, czy też za nim.

Aktualnie Twoim najważniejszym zadaniem jest podjęcie decyzji, co sprawia, że jednak piosenka jest większa od drugiej, oraz odpowiednie zaimplementowanie metody `compareTo()`. Wartość mniejsza od zera (dowolna) będzie oznaczać, że obiekt `Piosenka` przekazany w wywołaniu metody jest większy od obiektu, którego metoda została wywołana. Z kolei zwrócenie liczby większej od zera oznacza, że to obiekt, którego metodę wywołano, jest większy od obiektu przekazanego jako argument. Jeśli metoda `compareTo()` zwróci wartość zero, będzie to oznaczać, że obie piosenki są sobie równe (przynajmniej pod względem kryteriów sortowania... nie musi to wcale oznaczać, że jest to ten sam obiekt). Na przykład mogą istnieć dwie piosenki o tym samym tytule.

(A to z kolei może być przyczyną całej masy zupełnie nowych błędów i problemów, którymi zajmiemy się nieco później...).

**Oto kluczowe pytanie:
co sprawia, że jedna
piosenka jest mniejsza,
większa od innej piosenki
lub jej równa?**

**Nie będziesz mógł
zaimplementować
interfejsu Comparable,
póki tego nie określisz.**

Zaostrz ołówek



Napisz, jak według Ciebie powinna działać metoda `compareTo()`, by sortowała piosenki na podstawie tytułu. Napisz także pseudokod tej metody (albo jeszcze lepiej — jej PRAWDZIwy kod).

Podpowiedź: jeśli jesteś na dobrej drodze, to cała metoda powinna zająć nie więcej niż 3 wiersze kodu.

Nowa, poprawiona, porównywalna klasa Piosenka

Zdecydowaliśmy, że chcemy sortować piosenki według tytułu, dlatego też zaimplementowaliśmy metodę `compareTo()` w taki sposób, by porównywała tytuł piosenki przekazanej w wywołaniu metody z tytułem piosenki, której metoda została wywołana. Innymi słowy, to piosenka, której metoda została wywołana, musi zdecydować, jaki jest jej tytuł w porównaniu z tytułem piosenki przekazanej jako argument.

Hm... wiemy, że klasa `String` musi znać zasady określania kolejności alfabetycznej, gdyż lista łańcuchów znaków była sortowana prawidłowo. Wiemy, że klasa ta musi udostępnić metodę `compareTo()`... a zatem czemu nie mielibyśmy z niej skorzystać? W ten sposób wystarczy, że porównamy łańcuch znaków zawierający tytuł piosenki z tytułem drugiej piosenki i unikniemy konieczności pisania całego algorytmu porównywania i określania kolejności alfabetycznej.

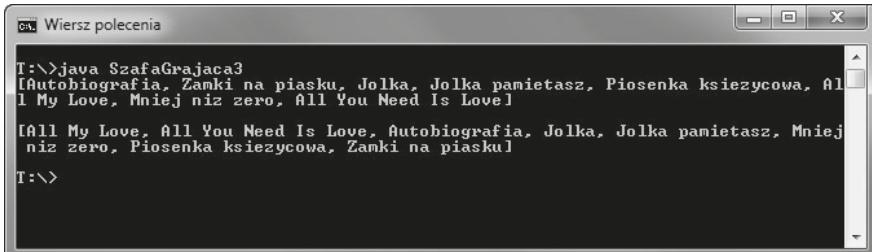
```
class Piosenka implements Comparable<Piosenka> {  
    String tytul;  
    String artysta;  
    String ocena;  
    String bpm;  
  
    public int compareTo(Piosenka p) {  
        return tytul.compareTo(p.getTytul());  
    }  
  
    Piosenka(String t, String a, String o, String b) {  
        tytul = t;  
        artysta = a;  
        ocena = o;  
        bpm = b;  
    }  
  
    public String getTytul() {  
        return tytul;  
    }  
  
    public String getArtysta() {  
        return artysta;  
    }  
  
    public String getOcena() {  
        return ocena;  
    }  
  
    public String.getBpm() {  
        return bpm;  
    }  
  
    public String toString() {  
        return tytul;  
    }  
}
```

Zauważaj te dwa typy są takie same... określamy typ, z jakim dana klasa będzie mogła być porównywana. To oznacza, że jeden obiekt `Piosenka` będzie można na potrzeby sortowania porównać z innym obiektem `Piosenka`.

Metoda `sort()` przekazuje obiekt `Piosenka` do metody `compareTo()`, by przekonać się, jaki ten obiekt jest w porównaniu z obiektem użyтыm do wywołania metody.

Proste! Całą robotę przekazujemy do łańcucha znaków zawierającego tytuł piosenki — wiemy bowiem, że klasa `String` posiada metodę `compareTo()`.

Tym razem wszystko zadziałało. Twój program najpierw wyświetla listę piosenek, a następnie wywołuje metodę `sort()`, która porządkuje je w kolejności alfabetycznej na podstawie tytułu.



Możemy już sortować listę, ale...

Masz nowy problem... Leon chciałby prezentować listę piosenek na dwa różne sposoby — posortowaną na podstawie tytułu bądź artysty!

Jednak kiedy zapewniałeś możliwość porównywania elementów kolekcji (poprzez zaimplementowanie interfejsu Comparable), mogłeś to zrobić tylko na jeden sposób — ponieważ metodę compareTo() mogłeś zaimplementować tylko raz. Co zatem możesz zrobić?

Dosyć makabrycznym rozwiążaniem byłoby dodanie do klasy Piosenka zmiennej (flagi) oraz warunku *if* do kodu metody compareTo(), który zmieniałby jej działanie w zależności od bieżącej wartości flagi — raz metoda porównywałaby piosenki na podstawie tytułu, a drugi — na podstawie nazwy artysty.

Jednak takie rozwiązanie jest okropne i podatne na występowanie błędów, a co ważniejsze... istnieje znacznie lepszy sposób. Coś, co zostało dodane do biblioteki Javy właśnie w tym celu — by pomóc Ci sortować obiekty na kilka różnych sposobów.

Jeszcze raz przyjrzyj się klasie Collections. Jest w niej dostępna także druga wersja metody sort() — umożliwiająca podanie komparatora.

To wciąż za mało. Czasami chciałbym sortować piosenki na podstawie nazwy artysty, a nie tytułu.



Metoda sort() została przeciążona, a jej druga wersja pozwala na przekazanie czegoś typu Comparator.

Uwaga dla mnie: muszę się dowiedzieć, jak tworzyć lub uzyskiwać ten „Comparator”, który pozwoliłby mi porównywać i sortować piosenki na podstawie nazwy artysty, a nie tytułu...

<code><T extends Comparable<? super T></code>	<code>void sort(List<T> list)</code>	Sorts the specified list into ascending order, according to the <i>natural ordering</i> of its elements.
<code><T> void</code>	<code>sort(List<T> list, Comparator<? super T> c)</code>	Sorts the specified list according to the order induced by the specified comparator.
<code>static void</code>	<code>swap(List<?> list, int i, int j)</code>	Swaps the elements at the specified positions in the specified list.
<code><I> Collection<I></code>	<code>synchronizedCollection(Collection<I> c)</code>	Returns a synchronized (thread-safe) collection backed by the specified collection.
<code>-</code>	<code>synchronizedList(List<I> list)</code>	

Stosowanie własnych komparatorów

Korzystając z metody `compareTo()`, element listy może porównywać się z innymi elementami tego samego typu tylko na jeden sposób. Jednak komparator jest niezależny od typu porównywanego elementu — to całkowicie niezależna klasa. Co więcej, można tworzyć dowolnie dużo takich klas! Chcemy porównywać piosenki na podstawie artysty? Wystarczy utworzyć klasę `ArtystaCompare`. Interesuje nas tempo utworów? Stwórzmy więc klasę `BPMCompare`.

Kiedy już stworzymy komparator, pozostało nam jedynie wywołać przeciążoną metodę `sort()`, która wymaga przekazania listy oraz obiektu `Comparator` pomagającego w posortowaniu zawartości listy w odpowiedniej kolejności.

Przeciążona wersja metody `sort()` podczas porównywania elementów wykorzysta komparator, a nie metodę `compareTo()`, zaimplementowaną w elementach listy. Innymi słowy, jeśli w wywołaniu metody `sort()` zostanie przekazany obiekt `Comparator`, to metoda ta nawet nie będzie wywoływać metody `compareTo()` elementów listy. Zamiast tego używana będzie metoda `compare()` obiektu `Comparator`.

A zatem oto nowe zasady:

- Wywołanie jednoargumentowej wersji metody `sort(List o)` oznacza, że kolejność elementów będzie określana przy użyciu metody `compareTo()` zaimplementowanej w elementach listy. Oznacza to, że elementy listy MUSZĄ implementować interfejs `Comparable`.
- Wywołanie przeciążonej metody `sort(List o, Comparator c)` oznacza, że metoda `compareTo()` elementów listy NIE będzie wywoływana, a zamiast niej zostanie użyta metoda `compare()` obiektu `Comparator`. To oznacza, że elementy listy NIE muszą implementować interfejsu `Comparable`.

Nie istnieją głupie pytania

P: Czy to oznacza, że jeśli mamy klasę, która nie implementuje interfejsu `Comparable`, i nie dysponujemy jej kodem źródłowym, to tworząc obiekt `Comparator`, i tak możemy posortować listę obiektów takiej klasy?

O: Owszem. Inne rozwiązanie (nie zawsze możliwe) może polegać na stworzeniu klasy potomnej elementu i zimplementowaniu w niej interfejsu `Comparable`.

P: Ale dlaczego nie wszystkie klasy implementują interfejs `Comparable`?

O: Czy naprawdę uważasz, że wszystko można posortować i uporządkować? Jeśli używasz elementów, których typy nie nadają się do uporządkowania w żaden naturalny, niewymuszony sposób, to implementując w nich interfejs `Comparable`, mógłbyś wprowadzić innych programistów w błąd. Poza tym, tworząc własne obiekty `Comparator`, można porównywać wszystko ze wszystkim; dlatego nie ponosisz żadnego ryzyka, jeśli w swojej klasie nie zdecydujesz się zaimplementować interfejsu `Comparable`.

java.util.Comparator

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

Przystosowanie Szafy muzycznej do korzystania z komparatora

W zamieszczonym poniżej kodzie wprowadziliśmy trzy zmiany:

1. Stworzyliśmy klasę wewnętrzną implementującą interfejs Comparator (czyli także metodę `compare()`, która wykonuje te same czynności, jakie wcześniej realizowała metoda `compareTo()`).
2. Stworzyliśmy obiekt wewnętrznej klasy Comparator.
3. Zastosowaliśmy przeciążoną wersję metody `sort()`, przekazując w jej wywołaniu zarówno sortowaną listę, jak i nasz obiekt Comparator.

Uwaga: dodatkowo zmodyfikowaliśmy także metodę `toString()` klasy Piosenka, by wyświetlała zarówno tytuł utworu, jak i nazwę artysty. (Aktualnie metoda ta wyświetla informacje w postaci `tytuł:artysta`, niezależnie od tego, w jaki sposób będziemy sortować listę).

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

public class SzafaGrajaca4 {

    ArrayList<Piosenka> listaPiosenek = new ArrayList<Piosenka>();

    public static void main(String[] args) {
        new SzafaGrajaca4().doDziela();
    }

    class ArtystaCompare implements Comparator<Piosenka> {
        public int compare(Piosenka p1, Piosenka p2) {
            return p1.getArtysta().compareTo(p2.getArtysta());
        }
    } To będzie tańcuch znaków (zawierający nazwę artysty).

    public void doDziela() {
        pobierzPiosenki();
        System.out.print(listaPiosenek);
        Collections.sort(listaPiosenek);
        System.out.println(listaPiosenek);

        ArtystaCompare komparator = new ArtystaCompare();
        Collections.sort(listaPiosenek, komparator);

        System.out.println(listaPiosenek);
    }

    void pobierzPiosenki() {
        // kod obsługujący pobieranie piosenek z pliku
    }

    void dodajPiosenke(String wierszDoAnalizy) {
        // kod analizujący wiersz tekstu i tworzący obiekt Piosenka
    }
}
```

Tu powinno być komentarz o tym, że zmieniono nazwy metod, aby nie konflikowały z nazwami zewnętrznych klas.

Tworzymy nową klasę wewnętrzną, implementującą interfejs Comparator (zwróć uwagę, że parametr typu tego interfejsu odpowiada typowi obiektów, które będziemy porównywać — w naszym przypadku są to obiekty Piosenka).

Pozwalamy, by to tańcuchy znaków same dokonały faktycznego porównania. Korzystamy tu z faktu, że tańcuchy znaków wiedzą, jak się porównywać i ustawać w kolejności alfabetycznej.

Tworzymy obiekt klasy wewnętrznej implementującej interfejs Comparator.

Wywołujemy metodę `sort()`, przekazując do niej listę oraz odwołanie do nowego obiektu typu Comparator.

Uwaga: w naszym przypadku sortowanie na podstawie tytułu jest domyślnym sposobem sortowania piosenek, gdyż metoda `compareTo()` klasy Piosenka operuje właśnie na tytułach. Inne rozwiązanie mogłoby polegać na zaimplementowaniu sortowania na podstawie tytułów oraz artystów jako dwóch odrębnych klas wewnętrznych implementujących interfejs Comparator, przy jednoczesnym zrezygnowaniu z implementacji interfejsu Comparable. W takim przypadku zawsze trzeba był sortować listę piosenek, używając dwuargumentowej wersji metody `sort()`.

Ćwiczenie wykorzystania kolekcji

```
import _____;

public class SortowanieGor {

    LinkedList_____ mtn = new LinkedList_____();

    class NazwaCompare _____ {
        public int compare(Gora g1, Gora g2) {
            return _____;
        }
    }

    class WysokoscCompare _____ {
        public int compare(Gora g1, Gora g2) {
            return (_____);
        }
    }

    public static void main(String [] args) {
        new SortowanieGor().doDziela();
    }

    public void doDziela() {
        mtn.add(new Gora("Kasprowy", 1987));
        mtn.add(new Gora("Koscielec", 2155));
        mtn.add(new Gora("Swinica", 2301));
        mtn.add(new Gora("Rysy", 2499));
        System.out.println("Bez sortowania: \n" + mtn);
        NazwaCompare nc = new NazwaCompare();
        ;
        System.out.println("Wg. nazwy:\n" + mtn);
        WysokoscCompare wc = new WysokoscCompare();
        ;
        System.out.println("Wg. wysokosci:\n" + mtn);
    }
}

class Gora {
    _____;
    _____;

    _____ {
        _____;
        _____;
    }

    _____ {
        _____;
        _____;
    }
}
```



Zaostrz ołówek Inżynieria wstępna

Przyjmij, że przedstawiony kod jest umieszczony w jednym pliku. Twoim zadaniem jest wypełnienie pustych miejsc kodu w taki sposób, by utworzony program wygenerował wyniki przedstawione u dołu strony.

Uwaga: Odpowiedzi zostały podane na końcu rozdziału.

```
Wiersz polecenia
T:>java SortowanieGor
Bez sortowania:
[Kasprowy 1987, Koscielec 2155, Swinica 2301, Rysy 2499]
Wg. nazwy:
[Kasprowy 1987, Koscielec 2155, Rysy 2499, Swinica 2301]
Wg. wysokosci:
[Rysy 2499, Swinica 2301, Koscielec 2155, Kasprowy 1987]
T:>>
```



Zaostrz ołówek

Wypełnij puste miejsca

Dla każdego z zamieszczonych poniżej pytań wypełnij puste miejsca, zapisując w nich jedno ze słów podanych na liście „możliwych odpowiedzi” i podając prawidłową odpowiedź na zadane pytanie. Odpowiedzi znajdziesz na końcu tego rozdziału.

Możliwe odpowiedzi:

Comparator,
Comparable,
compareTo(),
compare(),
tak,
nie.

Dysponując następującą instrukcją:

```
Collections.sort(mojaArrayList);
```

Odpowiedz na pytanie:

1. Jaki interfejs musi implementować klasa, której obiekty są przechowywane w kolekcji ArrayList? _____
2. Jaką metodę musi implementować klasa, której obiekty są przechowywane w kolekcji ArrayList? _____
3. Czy klasa obiektów przechowywanych w kolekcji ArrayList może jednocześnie implementować oba interfejsy — Comparator **oraz** Comparable? _____

Dysponując następującą instrukcją:

```
Collections.sort(mojaArrayList, mojKomparator);
```

Odpowiedz na pytanie:

4. Czy klasa obiektów przechowywanych w kolekcji ArrayList może implementować interfejs Comparable? _____
5. Czy klasa obiektów przechowywanych w kolekcji ArrayList może implementować interfejs Comparator? _____
6. Czy klasa obiektów przechowywanych w kolekcji ArrayList musi implementować interfejs Comparable? _____
7. Czy klasa obiektów przechowywanych w kolekcji ArrayList musi implementować interfejs Comparator? _____
8. Jaki interfejs musi implementować klasa obiektu przekazanego jako mojKomparator? _____
9. Jaką metodę musi implementować klasa obiektu przekazanego jako mojKomparator? _____

Usuwanie powtórzeń

O rany. Sortowanie działa w porządku, ale teraz pojawiły się powtórzenia...

Sortowanie działa doskonale; teraz już wiemy, jak sortować zarówno po *tytule* (wykorzystując w tym celu metodę *compareTo()* klasy *Piosenka*), jak i po *artyście* (używając metody *compare()* obiektu *Comparator*). Jednak pojawił się nowy problem, którego nie zauważylismy podczas pisania aplikacji, gdy używaliśmy testowego pliku z piosenkami generowanego przez szafę grającą. Problem polega na tym, że nasza posortowana lista piosenek zawiera powtórzenia.

Wygląda na to, że szafa grająca zawsze zapisuje piosenki do pliku, niezależnie od tego, czy były one już wcześniej odtwarzane (czyli zapisane w pliku), czy też nie. Plik tekstowy *PelnaListaPiosenek.txt* generowany przez szafę grającą jest kompletnym zapisem wszystkich piosenek, jakie zostały odtworzone, i może zawierać wiele wpisów dla tego samego utworu.

```
T:\>java SzafaGrajaca4
[Jolka, Jolka pamietasz: Budka Suflera, Piosenka ksiezycowa: VARIUS MANX, Mniej niz zero: Lady Pank, Zamki na piasku: Lady Pank, All You Need Is Love: The Beatles, All My Love: LED ZEPPELIN, Autobiografia: Perfect, Autobiografia: Perfect, Autobiografia: Perfect, Zamki na piasku: Lady Pank]
[All My Love: LED ZEPPELIN, All You Need Is Love: The Beatles, Autobiografia: Perfect, Autobiografia: Perfect, Autobiografia: Perfect, Jolka, Jolka pamietasz: Budka Suflera, Mniej niz zero: Lady Pank, Piosenka ksiezycowa: VARIUS MANX, Zamki na piasku: Lady Pank, Zamki na piasku: Lady Pank]
[Jolka, Jolka pamietasz: Budka Suflera, All My Love: LED ZEPPELIN, Mniej niz zero: Lady Pank, Zamki na piasku: Lady Pank, Zamki na piasku: Lady Pank, Autobiografia: Perfect, Autobiografia: Perfect, Autobiografia: Perfect, All You Need Is Love: The Beatles, Piosenka ksiezycowa: VARIUS MANX]
T:\>_
```

Przed posortowaniem.

Po posortowaniu przy użyciu metody *compareTo()* klasy *Piosenka* (czyli sortowaniu na podstawie tytułu).

Po posortowaniu przy użyciu komparatora *ArtystaCompare* (czyli sortowaniu na podstawie nazwy artysty).

PelnaListaPiosenek.txt

```
Jolka, Jolka pamietasz/Budka Suflera/9/100
Piosenka ksiezycowa/VARIUS MANX/7/90
Mniej niz zero/Lady Pank/7/100
Zamki na piasku/Lady Pank/5/120
All You Need Is Love/The Beatles/10/90
All My Love/LED ZEPPELIN/9/120
Autobiografia/Perfect/7/80
Autobiografia/Perfect/7/80
Autobiografia/Perfect/7/80
Zamki na piasku/Lady Pank/5/120
```

Aktualnie plik tekstowy *PelnaListaPiosenek.txt* zawiera powtórzenia, gdyż nowa szafa grająca zapisuje w nim każdą odtwarzaną piosenkę w takiej kolejności, w jakiej były odtwarzane. Jak widać, ktoś puścił piosenkę „Autobiografia” trzy razy pod rząd, a następnie piosenkę „Zamki na piasku”, która także była już wcześniej odtwarzana. Nie możemy zmienić sposobu generowania pliku tekowego, gdyż czasami będziemy potrzebowali wszystkich tych informacji. Musimy zatem zmienić kod naszej aplikacji.

Potrzebujemy zbioru, a nie listy

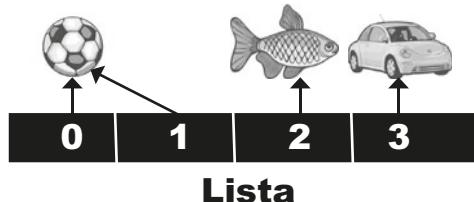
W bibliotece kolekcji Javy możemy znaleźć trzy podstawowe interfejsy — List, Set oraz Map. Używana przez nas kolekcja ArrayList implementuje interfejs List, jednak wygląda na to, że to interfejs Set (zbior) jest właśnie tym, czego nam potrzeba.

Powtórzenia są OK.

➤ LIST (lista) — kiedy kolejność ma znaczenie

Kolekcje, które znają **położenie** elementu (jego **indeks**).

Listy wiedzą, w jakim miejscu znajdują się ich poszczególne elementy. Może się na nich znaleźć kilka elementów odwołujących się do tego samego obiektu.

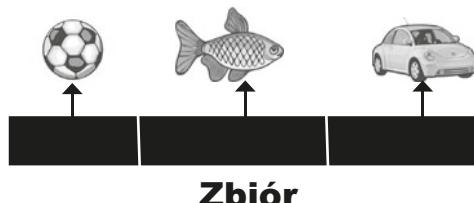


➤ SET (zbior) — gdy ważna jest unikalność

Te kolekcje **nie pozwalają** na występowanie **powtarzających się elementów**.

Zbiorzy wiedzą, czy konkretnie elementy już należą do kolekcji, czy jeszcze ich w nich nie ma. W zbiorze nigdy nie może się znaleźć więcej niż jeden element odwołujący się do tego samego obiektu (albo więcej niż jeden element odwołujący się do dwóch obiektów uważanych za równe — tą relacją równości zajmiemy się już niebawem).

ŻADNYCH powtórzeń.

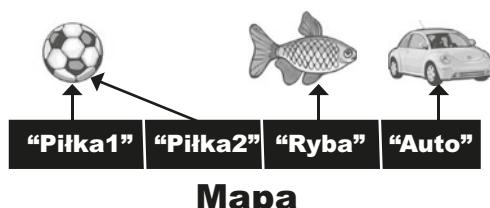


➤ MAP (mapa) — Gdy ważne jest odnajdywanie elementów na podstawie klucza

Te kolekcje operują na **parach klucz-wartość**.

Mapy znają wartości skojarzone z podanym kluczem. Nic nie stoi na przeszkodzie, by z jedną wartością skojarzyć kilka kluczy, jednak same klucze nie mogą się powtarzać. Choć kluczami zazwyczaj są łańcuchy znaków (dzięki temu można na przykład stworzyć listę właściwości — nazwa-wartość), to jednak mogą nimi być także obiekty.

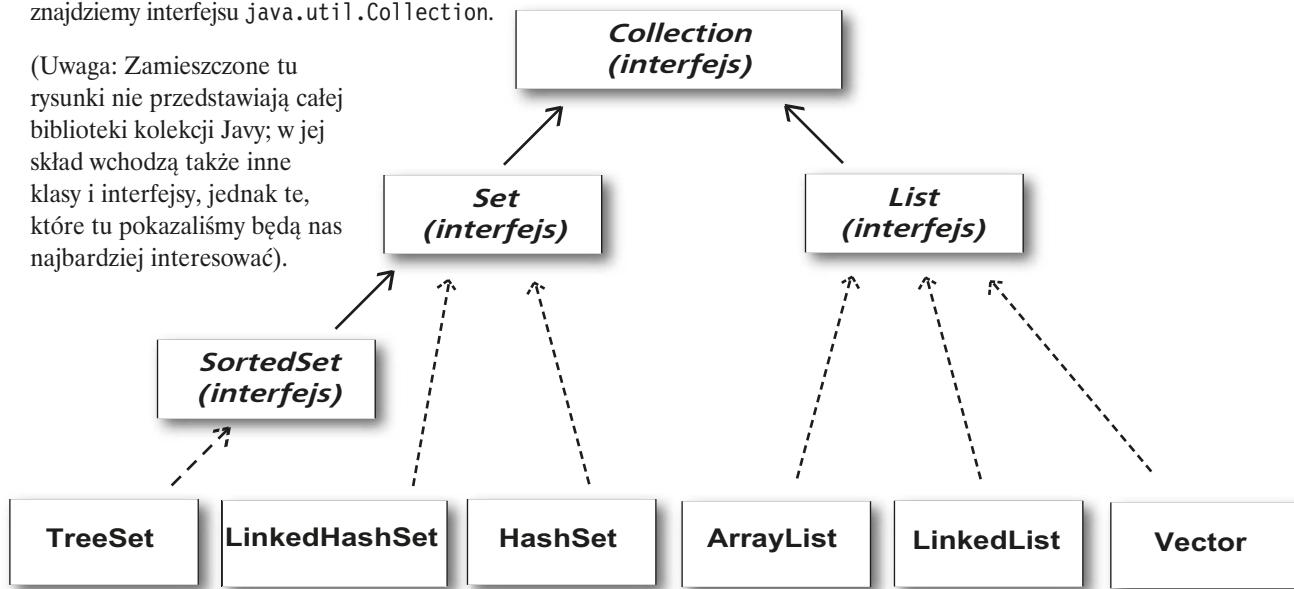
Wartości mogą się powtarzać, ale klucze NIE.



Biblioteka kolekcji (fragment)

Zauważ, że interfejs Map nie rozszerza interfejsu Collection, a pomimo to i tak jest uważany za element „biblioteki kolekcji” (nazywanej także *Collection API*). A zatem mapy wciąż są kolekcjami, choć w ich drzewie dziedziczenia nie znajdziemy interfejsu java.util.Collection.

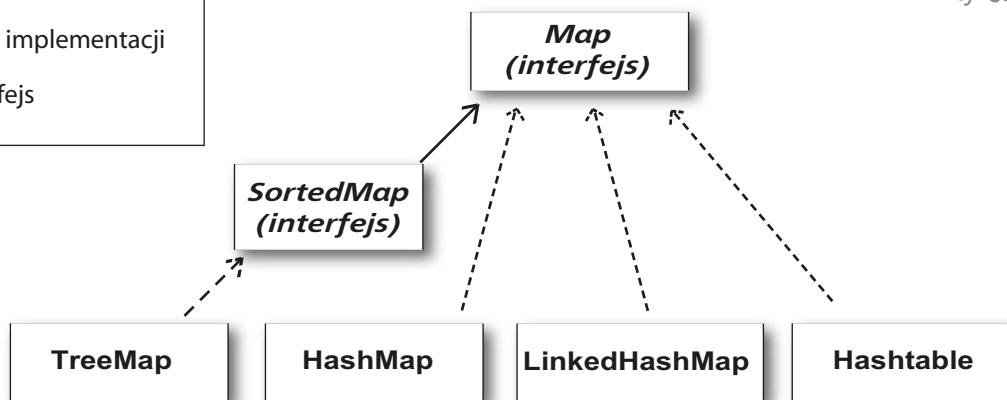
(Uwaga: Zamieszczone tu rysunki nie przedstawiają całej biblioteki kolekcji Javy; w jej skład wchodzą także inne klasy i interfejsy, jednak te, które tu pokazaliśmy będą nas najbardziej interesować).



Legenda

	rozszerza
	implementuje
	klasa implementacji
	interfejs

Mapy nie implementują interfejsu java.util.Collection, jednak i tak są uważane za element biblioteki kolekcji Javy.



Wykorzystanie kolekcji HashSet zamiast ArrayList

Zmodyfikowaliśmy nieco naszą aplikację i zastosowaliśmy w niej kolekcję HashSet. (Uwaga: pomineliśmy w niej niektóre fragmenty kodu, jednak możesz je bezpośrednio skopiować z poprzedniej wersji programu. Poza tym, aby łatwiej było analizować generowane wyniki, wróciliśmy do wcześniejszej wersji metody `toString()` klasy Piosenka, która wyświetlała jedynie tytuł utworu, a nie tytuł i wykonawcę).

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

public class SzafaGrajaca6 {

    ArrayList<Piosenka> listaPiosenek = new ArrayList<Piosenka>(); ←

    // metoda main
    public void doDziela() { ← Nie zmienialiśmy metody pobierzPiosenki(), więc
        pobierzPiosenki(); ← piosenki odczytane z pliku tekstowego wciąż są
        System.out.print(listaPiosenek); ← zapisywane w kolekcji ArrayList.
        Collections.sort(listaPiosenek); ←
        System.out.println(listaPiosenek); ←

        HashSet<Piosenka> zbiorPiosenek = new HashSet<Piosenka>(); ← Tutaj tworzymy nową kolekcję HashSet
        zbiorPiosenek.addAll(listaPiosenek); ← sparametryzowaną do przechowywania
        System.out.println(zbiorPiosenek); ← obiektów Piosenka.

    } ← Kolekcja HashSet posiada prostą metodę addAll(), która
        // metody: pobierzPiosenki() oraz dodajPiosenke(String wierszDoAnalizy)

}
```

Przed sortowaniem listy piosenek.

Po posortowaniu kolekcji ArrayList (według tytułu piosenki).

Po zapisaniu piosenek w kolekcji HashSet i wyświetleniu jego zawartości (nie wywoływaliśmy powtórnie metody `sort()`).

Zastosowanie zbioru wcale nam nie pomogło!! Piosenki wciąż się powtarzają!

(Poza tym po umieszczeniu elementów w zbiorze zmienią one swoją kolejność, ale tym problemem zajmiemy się później...)

Co sprawia, że dwa obiekty są sobie równe?

W pierwszej kolejności musimy sobie zadać pytanie — na jakiej podstawie możemy uznać, że dwa odwołania do obiektów Piosenka stanowią powtórzenie? Otóż muszą one zostać uznane za **równe**. Czy muszą to być dwa odwołania do tego samego obiektu, czy też mogą to być dwa różne obiekty Piosenka mające ten sam *tytuł*?

I w ten sposób doszliśmy do kluczowego zagadnienia — czy chodzi nam o równość *odwołań*, czy też o równość obiektów.

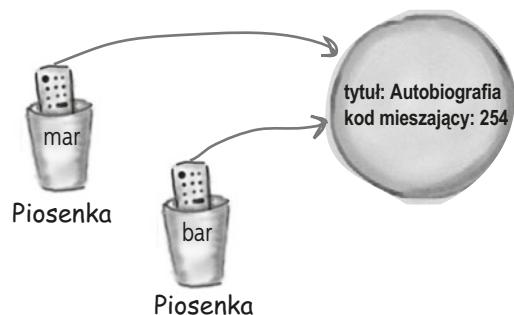
➤ Równość odwołań

Dwa odwołania, jeden obiekt na stercie.

Dwa odwołania wskazujące ten sam obiekt na stercie są sobie równe. I kropka. Jeśli wywołasz metodę `hashCode()` obu zmiennych referencyjnych, uzyskasz takie same wartości. Jeśli nie przesłoniś tej metody, to jej domyślne działanie (pamiętaj, że każdy obiekt dziedziczy ją z klasy `Object`) będzie polegało na zwracaniu unikalnych wartości dla wszystkich obiektów (większość wersji Javy zwraca kod mieszający określany na podstawie adresu obiektu na stercie, dzięki temu żadne dwa obiekty nie będą miały tego samego kodu).

Jeśli chcesz wiedzieć, czy dwa *odwołania* wskazują na ten sam obiekt, użyj operatora `==`, który (jak zapewne pamiętasz) porównuje bity odwołań. Jeśli dwa odwołania wskazują ten sam obiekt, to ich bity będą identyczne.

Jeśli dwa obiekty `mar` i `bar` są równe, wywołanie `mar.equals(bar)` musi zwracać wartość `true`, a poza tym wartości zwarcane przez metodę `hashCode()` obu tych obiektów muszą być identyczne. Aby zbiory uznały, że dwa obiekty są identyczne, musimy zatem przesłonić metody `hashCode()` oraz `equals()` odziedziczone od klasy `Object`. W ten sposób zapewniamy, że dwa różne obiekty zostaną uznane za równe.



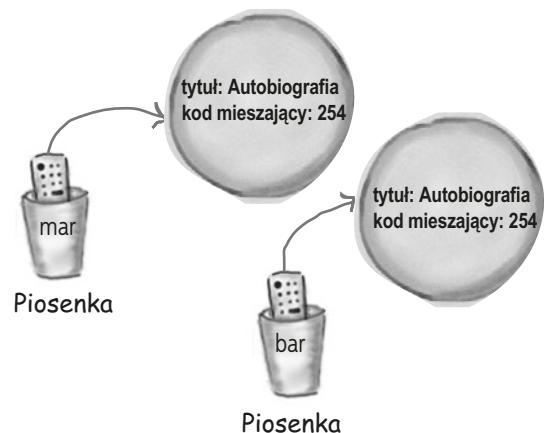
```
if (mar == bar) {  
    // oba odwołania wskazują ten  
    // sam obiekt na stercie  
}
```

➤ Równość obiektów

Dwa odwołania, dwa obiekty na stercie, jednak pod względem znaczenia obiekty można uznać za **tożsame**.

Jeśli chcesz traktować dwa różne obiekty Piosenka jako równe (na przykład, jeśli uznasz, że piosenki są równe, jeśli mają identyczne *tytuły*), musisz przesłonić **dwie** metody — `hashCode()` oraz `equals()` — dziedziczone z klasy `Object`.

Jak już zaznaczyliśmy wcześniej, jeśli *nie* przesłoniś metody `hashCode()`, to domyślnie będzie ona zwracać unikalny kod mieszający dla każdego obiektu. Dlatego musisz ją przesłonić, by zagwarantować, że dwa tożsame obiekty będą miały identyczne kody. Jednocześnie jednak musisz przesłonić metodę `equals()`, by zawsze zwracała wartość **true**, jeśli metoda zostanie wywołana na rzecz *któregokolwiek* z tych obiektów, a drugi zostanie przekazany jako argument.



```
if (mar.equals(bar) && mar.hashCode() == bar.hashCode()) {  
    // obie referencje odwołują się bądź do tego samego obiektu,  
    // bądź dwóch obiektów, które są sobie równe.  
}
```

W jaki sposób kolekcja HashSet szuka powtórzeń: metody hashCode() oraz equals()

Kiedy umieszczasz obiekt w kolekcji HashSet, używa ona kodu mieszającego obiektu, by określić, w jakim miejscu zbioru należy go umieścić. Jednocześnie kod mieszający dodawanego obiektu jest porównywany z kodami wszystkich innych obiektów, które już znajdują się w zbiorze — jeśli kod mieszający nowego obiektu nie zostanie odnaleziony, to kolekcja zakłada, że nowy obiekt nie jest powtórzeniem i można go dodać.

Innymi słowy, jeśli kody są różne, to kolekcja HashSet zakłada, że obiekty nie mogą być sobie równe!

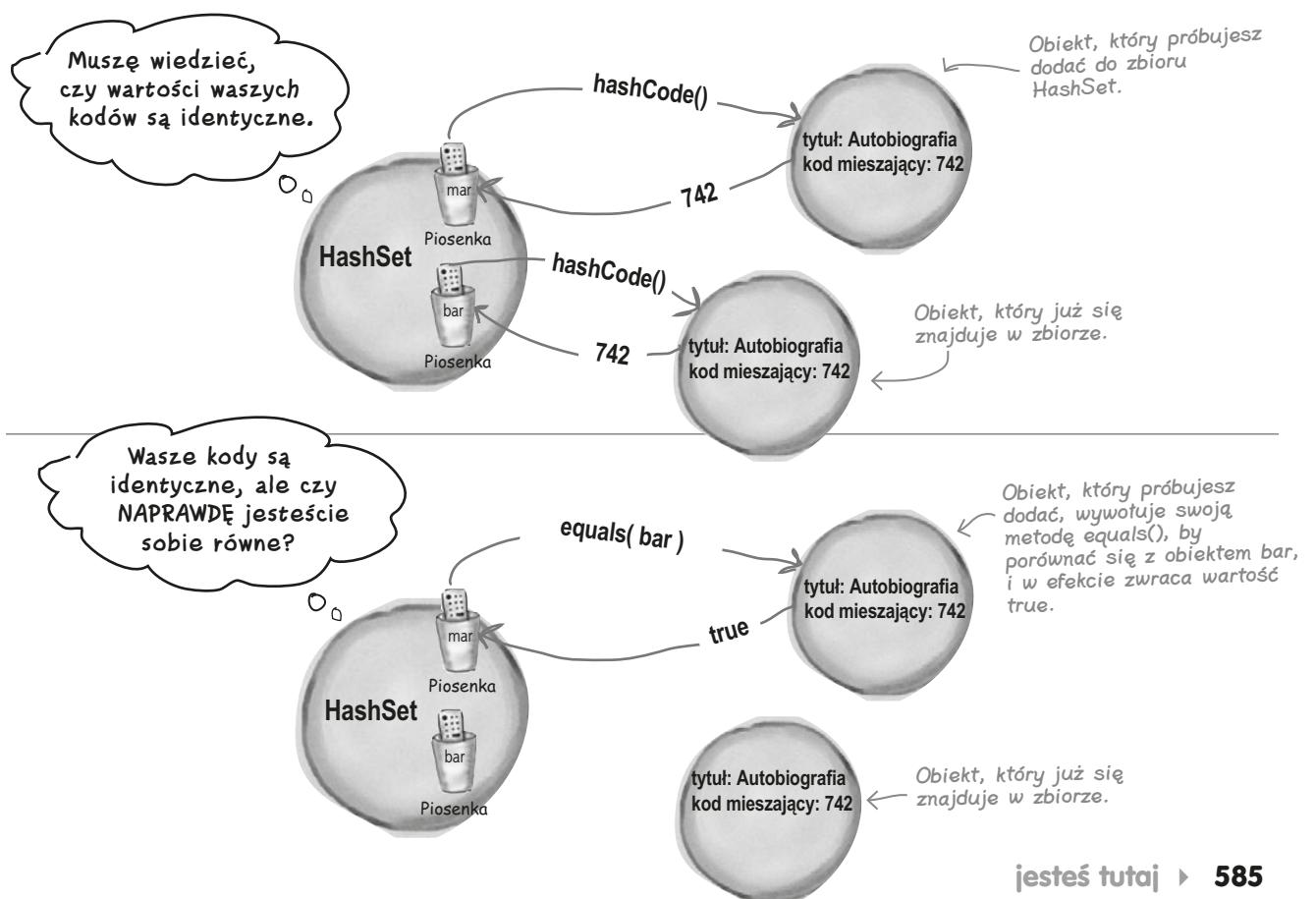
Dlatego musisz przesłonić metodę hashCode(), by zagwarantować, że obiekty równe sobie będą miały tę samą wartość kodu mieszającego.

Jednak dwa obiekty o tej samej wartości kodu mieszającego niekoniecznie muszą być sobie równe (więcej informacji na ten

temat znajdziesz na następnej stronie). Dlatego też, jeśli okaże się, że kody mieszające dwóch obiektów są identyczne, kolekcja HashSet wywoła dodatkowo metodę equals() jednego z tych obiektów, by upewnić się, że naprawdę są one równe.

Jeśli oba obiekty są równe, to HashSet będzie wiedzieć, że próbujesz dodać obiekt będący duplikatem innego obiektu, już przechowywanego w zbiorze, i przerwie całą operację.

W takim przypadku nie zostanie zgłoszony żaden wyjątek, lecz metoda add() zwróci wartość logiczną informującą (jeśli to kogoś obchodzi), czy udało się dodać obiekt do zbioru, czy nie. A zatem, jeśli metoda add() zwróci wartość false, będziesz wiedzieć, że dodawany obiekt stanowił powtórzenie obiektu już przechowywanego w kolekcji.



Klasa Piosenka z przesłoniętymi metodami hashCode() i equals()

```
class Piosenka implements Comparable<Piosenka> {

    String tytul;
    String artysta;
    String ocena;
    String bpm;

    public boolean equals(Object piosenka) {
        Piosenka p = (Piosenka) piosenka;
        return getTytul().equals(p.getTytul());
    }

    public int hashCode() {
        return tytul.hashCode();
    }

    public int compareTo(Piosenka s) {
        return tytul.compareTo(s.getTytul());
    }

    Piosenka(String t, String a, String o, String b) {
        tytul = t;
        artysta = a;
        ocena = o;
        bpm = b;
    }

    public String getTytul() {
        return tytul;
    }

    public String getArtysta() {
        return artysta;
    }

    public String getOcena() {
        return ocena;
    }

    public String.getBpm() {
        return bpm;
    }

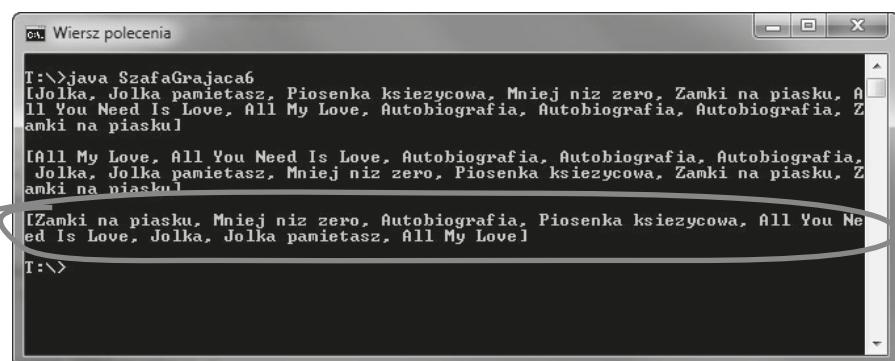
    public String toString() {
        return tytul;
    }
}
```

Zbiór HashSet (lub jakikolwiek inny obiekt wywołujący tę metodę) przekazuje do niej inny obiekt Piosenka.

To bardzo WAŻNA wiadomość — tytuł jestłańcuchem znaków, obiektem String, a obiekty String mają przesłoniętą metodę equals(). A zatem wszystko, co musimy zrobić, sprawdza się do zapytania, czy tytuł jednej piosenki jest równy tytułu drugiej piosenki.

Podobnie postępujemy także i tutaj. Ponieważ w klasie String została przesłonięta metoda hashCode(), zatem wystarczy, że zwrócimy wynik wywołania tej metody dlałańcucha znaków określającego tytuł piosenki. Zauważ, że obie metody, hashCode() oraz equals(), operują na TEJ SAMEJ składowej.

Teraz wszystko działa! Po wyświetleniu zbioru HashSet widać, że jego elementy nie powtarzają się. Wciąż jednak nie wywołaliśmy metody sort(), a w trakcie zapisywania zawartości kolekcji ArrayList w zbiorze kolejność jej elementów nie została zachowana.



Prawo obiektowe Javy dotyczące metod hashCode() i equals()

Dokumentacja klasy Object narzuca poniższe reguły, których MUSISZ przestrzegać:

- **Jeśli dwa obiekty są równe, to ich kody mieszające MUSZĄ być identyczne.**
- **Jeśli dwa obiekty są równe, wywołanie metody equals() każdego z nich MUSI zwrócić wartość true. Innymi słowy, jeśli a.equals(b), to b.equals(a).**
- **Jeśli dwa obiekty mają taką samą wartość kodu mieszającego, to NIE muszą być sobie równe. Jeśli jednak obiekty są równe, to ich wartości kodu mieszającego MUSZĄ być identyczne.**
- **A zatem, jeśli przesłaniasz metodę equals(), to MUSISZ także przesłonić metodę hashCode().**
- **Domyślne działanie metody hashCode() polega na wygenerowaniu unikalnej liczby całkowitej dla każdego obiektu na stercie. Jeśli zatem nie przesłonisz metody hashCode(), to dwa obiekty danej klasy NIGDY nie zostaną uznane za równe.**
- **Domyślne działanie metody equals() polega na wykorzystaniu operatora porównania ==, czyli sprawdzeniu, czy dwa odwołania wskazują na ten sam obiekt na stercie. A zatem, jeśli nie przesłonisz metody equals() w swojej klasie, to żadne dwa obiekty NIGDY nie zostaną uznane za równe, gdyż wzorce bitowe odwołań do dwóch różnych obiektów nigdy nie będą identyczne.**
- a.equals(b) musi także oznaczać, że a.hashCode() == b.hashCode().**
- Jednak spełnienie warunku a.hashCode() == b.hashCode() NIEkoniecznie musi oznaczać, że a.equals(b).**

I Nie istnieją grupie pytania

P: Jak to możliwe, że kody mieszające dwóch obiektów mogą być identyczne, nawet jeśli same obiekty nie są równe?

O: Zbiór HashSet używa kodów mieszających, by przechowywać obiekty w kolekcji w taki sposób, by można się było do nich możliwie jak najszybciej odwołać. Jeśli spróbujesz wyszukać obiekt w kolekcji ArrayList, przekazując w tym celu kopię poszukiwanego obiektu (a nie jego indeks), to kolekcja będzie musiała przejrzeć całą swoją zawartość element po elemencie. Z kolei zbiór HashSet jest w stanie odszukać obiekt znacznie szybciej, gdyż używa on kodu mieszającego obiektu jako swoistej „etykiety” do „kubelka”, w którym dany obiekt jest przechowywany. A zatem, jeśli zażadasz: „znajdź w swojej zawartości obiekt, który jest dokładnie taki sam jak ten...”, to zbiór HashSet pobierze kod mieszający przekazanego obiektu (w naszym przypadku może to być obiekt Pisanka, którego kod ma wartość #742), a następnie stwierdzi: „Ha... doskonale wiem, gdzie jest przechowywany obiekt o kodzie #742”, i skieruje się prosto do kubelka #742.

Nie są to zapewne równie wyczerpujące informacje jak te, które byś zdobył na kursie komputerowym, jednak powinny Ci wystarczyć, byś mógł efektywnie korzystać ze zbiorów HashSet. W rzeczywistości opracowanie dobrego algorytmu generowania kodów mieszających obiektów jest tematem wielu prac doktorskich i znacznie wykracza poza zagadnienia, które chcemy przedstawić w tej książce.

Najważniejsze jest, byś wiedział, że równość kodów nie oznacza wcale równości obiektów, gdyż może się zdarzyć, że „algorytm mieszający” (ang. *hashing algorithm*) będzie zwracał tę samą wartość dla wielu różnych obiektów. Owszem, oznacza to, że różne obiekty umieszczane w zbiorze HashSet mogą trafić do tego samego kubelka (gdyż każdy kubek reprezentuje unikalną wartość kodu mieszającego), niemniej jednak nie jest to równoznaczne z końcem świata. Może to oznaczać jedynie nieznaczne zmniejszenie wydajności działania zbioru (bądź fakt zapisania w nim niezwykle wielkiej liczby elementów). Jeśli jednak okaże się, że w kubelku dla danego kodu mieszającego znajduje się kilka obiektów, zbiór wykorzysta metodę equals(), by sprawdzić, czy jest wśród nich taki, który dokładnie odpowiada poszukiwanemu. Innymi słowy, wartości kodów są czasami używane do zawężania pola poszukiwań, jednak by odszukać dokładnie ten obiekt, o jaki nam chodzi, HashSet i tak musi sprawdzić wszystkie obiekty z danego kubelka (zawierającego obiekty o tym samym kodzie mieszającym) i dla każdego z nich wywołać metodę equals().

A jeśli chcemy, by zbiór zachował kolejność sortowania, mamy do dyspozycji kolekcję TreeSet

Kolekcje TreeSet oraz HashSet mają tę wspólną cechę, że obie nie dopuszczają do przechowywania takich samych obiektów. Jednak TreeSet dba dodatkowo o to, by elementy *zawsze były* odpowiednio posortowane. Zbiór ten działa podobnie jak metoda `sort()` — jeśli zostanie utworzony przy użyciu konstruktora bezargumentowego, to zapisane w nim obiekty będą sortowane przy użyciu zaimplementowanej w nich metody `comapreTo()`. Istnieje także możliwość przekazania do konstruktora klasy TreeSet obiektu Comparator — w takim przypadku to właśnie on będzie używany podczas sortowania zawartości zbioru.

Kolekcja ta ma jedną wadę — otóż w sytuacjach, gdy sortowanie nie jest nam *potrzebne*, i tak będzie wykonywane, narażając nas na niepotrzebne obniżenie wydajności. Jednak najprawdopodobniej sam stwierdzisz, że w przeważającej większości aplikacji to obniżenie wydajności związane z sortowaniem zawartości zbioru jest praktycznie niezauważalne.

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

public class SzafaGrajaca8 {

    ArrayList<Piosenka> listaPiosenek = new ArrayList<Piosenka>();

    public static void main(String[] args) {
        new SzafaGrajaca8().doDziela();
    }

    public void doDziela() {
        pobierzPiosenki();
        System.out.print(listaPiosenek);
        Collections.sort(listaPiosenek);
        System.out.println(listaPiosenek);

        TreeSet<Piosenka> zbiorPiosenek = new TreeSet<Piosenka>();
        zbiorPiosenek.addAll(listaPiosenek);
        System.out.println(listaPiosenek);
    }

    void pobierzPiosenki() {
        try {
            File plik = new File("ListaPiosenek.txt");
            BufferedReader reader = new BufferedReader(new FileReader(plik));
            String wiersz = null;
            while ((wiersz= reader.readLine()) != null) {
                dodajPiosenke(wiersz);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void dodajPiosenke(String wierszDoAnalizy) {
        String[] elementy = wierszDoAnalizy.split("/");
        Piosenka nastepnaPiosenka = new Piosenka(elementy[0], elementy[1], elementy[2], elementy[3]);
        listaPiosenek.add(nastepnaPiosenka);
    }
}
```

Tworzymy obiekt klasy TreeSet, a nie HashSet. Wykorzystanie konstruktora bezargumentowego oznacza, że do sortowania zbioru zostanie użyta metoda compareTo()
obiektów Piosenka.
(Moglibyśmy też przekazać w wywołaniu konstruktora obiekt Comparator).

Wszystkie piosenki z tablicy możemy dodać do zbioru przy użyciu metody addAll(). (Ewentualnie moglibyśmy także dodawać kolejno poszczególne piosenki, używając metody zbiorPiosenek.add(), tak samo jak dodawaliśmy je do kolekcji ArrayList).

Co MUSIMY wiedzieć o klasie TreeSet...

Można sądzić, że stosowanie klasy TreeSet jest łatwe, upewnijmy się jednak, czy naprawdę rozumiesz, co trzeba zrobić, by móc jej używać. Według nas jest to tak ważne zagadnienie, że postanowiliśmy zrobić z niego ćwiczenie, byś *musiał* dobrze się nad nim zastanowić.
NIE odwracaj kartki, zanim nie skończysz ćwiczenia. *Mówimy serio!*



Zaostrz ołówek

Spójrz na kod programu przedstawiony w ramce obok. Przeanalizuj go uważnie a następnie odpowiedz na pytania zamieszczone u dołu strony. (Uwaga: w kodzie programu nie ma żadnych błędów składniowych).

```
import java.util.*;

public class ZbiorTestowy {
    public static void main (String[] args) {
        new ZbiorTestowy().doDziela();
    }

    public void doDziela() {
        Ksiazka b1 = new Ksiazka("Jak działają koty");
        Ksiazka b2 = new Ksiazka("Remiks organiczny");
        Ksiazka b3 = new Ksiazka("Szukając Emo");
        TreeSet<Ksiazka> tree = new TreeSet<Ksiazka>();
        tree.add(b1);
        tree.add(b2);
        tree.add(b3);
        System.out.println(tree);
    }
}

class Ksiazka {
    String tytul;
    public Ksiazka(String t) {
        title = t;
    }
}
```

1. Co się stanie, kiedy spróbujesz skompilować ten program?

2. Jeśli uda się go skompilować, to jaki będzie rezultat jego wykonania?

3. Jeśli w przedstawionym kodzie jest jakiś problem (bądź to związany z komplikacją, bądź z działaniem), to jak byś go poprawił?

Elementy kolekcji TreeSet MUSZĄ być porównywalne

Kolekcja TreeSet nie potrafi czytać w myślach programisty i określić, jak należy porównywać zapisywane w niej obiekty. To Ty musisz jej *to* powiedzieć.

Aby móc używać kolekcji TreeSet, musi być spełnione jeden z poniższych warunków:

- **Elementami kolekcji muszą być obiekty typu implementującego interfejs Comparable.**

Klasa Ksiazka użyta w ostatnim przykładzie nie implementowała interfejsu Comparable, a zatem podczas próby wykonania programu zostałby zgłoszony wyjątek. Sam się nad tym zastanów... podstawowy celem istnienia kolekcji TreeSet jest przechowywanie elementów posortowanych w jakiejś kolejności — ale przecież kolekcja nie wie, jak należy sortować książek! Problem nie został zgłoszony podczas komplikacji, gdyż metoda add() kolekcji TreeSet nie wymaga przekazywania argumentów typu Comparable, lecz typu określonego podczas tworzenia kolekcji. Innymi słowy, jeśli utworzymy kolekcję przy użyciu wyrażenia new TreeSet<Ksiazka>, to metoda add() stanie się tak naprawdę metodą add(Ksiazka). A przecież nie ma żadnego wymogu, by klasa Ksiazka implementowała interfejs Comparable! Problem pojawia się w czasie wykonywania programu, podczas próby dodania do kolekcji drugiego elementu. Właśnie w tym momencie kolekcja po raz pierwszy próbuje wywołać metodę compareTo() jednego z obiektów... i okazuje się, że nie może tego zrobić!

LUB

- **Skorzystamy z przeciążonego konstruktora klasy TreeSet, umożliwiającego przekazanie obiektu typu Comparator.**

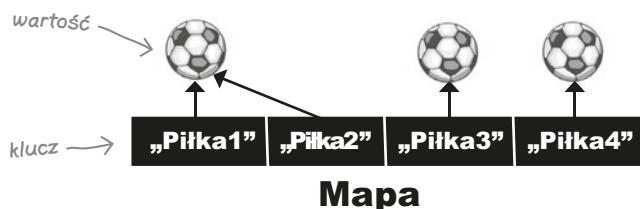
Kolekcja TreeSet działa podobnie do metody sort() — możesz użyć metody compareTo() porównywanych elementów (zakładając, że są to elementy klasy implementującej interfejs Comparable) ALBO stworzyć własny komparator, który zostanie następnie użyty do posortowania elementów kolekcji. By skorzystać z tej drugiej opcji, należy wywołać konstruktor klasy TreeSet umożliwiający przekazanie obiektu Comparator.

```
class Ksiazka implements Comparable {  
    String tytul;  
    public Ksiazka(String t) {  
        tytul = t;  
    }  
  
    public int compareTo(Object b) {  
        Ksiazka k = (Ksiazka) b;  
        return (tytul.compareTo(k.tytul));  
    }  
  
}  
  
public class KsiazkaCompare implements Comparator<Ksiazka> {  
    public int compare(Ksiazka k1, Ksiazka k2) {  
        return (k1.tytul.compareTo(k2.tytul));  
    }  
}  
  
public class Test {  
    // metoda main  
  
    public void doDziela() {  
        Ksiazka b1 = new Ksiazka("Jak działają koty");  
        Ksiazka b2 = new Ksiazka("Remiks organiczny");  
        Ksiazka b3 = new Ksiazka("Szukając Emo");  
        KsiazkaCompare komparator = new KsiazkaCompare();  
        TreeSet<Ksiazka> tree = new TreeSet<Ksiazka>(komparator);  
        tree.add(b1);  
        tree.add(b2);  
        tree.add(b3);  
        System.out.println(tree);  
    }  
}
```

Poznaliśmy już listy i zbiory, teraz poznamy mapy

Listy i zbiory są super, ale czasami okazuje się, że najlepszymi kolekcjami są mapy (choć także są one zaliczane do biblioteki kolekcji, to jednak klasy map nie implementują interfejsu Collection).

Wyobraź sobie, że chciałbyś mieć do dyspozycji kolekcję działającą jak lista właściwości — kolekcję, której podajesz nazwę i otrzymujesz wartość skojarzoną z tą nazwą. Choć tymi nazwami, określonymi w terminologii fachowej jako *klucz*, zazwyczaj są łańcuchy znaków, to jednak rolę klucza może pełnić dowolny obiekt Javy (oraz wartość typu podstawowego, dzięki automatycznej konwersji na obiekty).



Każdy element mapy składa się w rzeczywistości z dwóch obiektów — klucza i wartości. Mapa może zawierać powtarzające się wartości, ale NIE klucze.

Przykład zastosowania mapy

```
import java.util.*;  
  
public class TestMap {  
  
    public static void main(String[] args) {  
  
        HashMap<String, Integer> wyniki = new HashMap<String, Integer>();  
  
        wyniki.put("Kasia", 42);  
        wyniki.put("Berta", 343);  
        wyniki.put("Sabina", 420);  
        System.out.println(wyniki);  
        System.out.println(wyniki.get("Berta"));  
    }  
}
```

Kolekcja `HashMap` wymaga określenia dwóch parametrów typu — jednego dla kluczy i drugiego dla wartości.

W tym przypadku używamy metody `put()` zamiast `add()`, no i oczywiście musimy przekazać dwa argumenty (klucz i wartość).

Metoda `get()` wymaga podania klucza i zwraca skojarzoną z nim wartość (w tym przypadku będzie to obiekt `Integer`).

Wiersz polecenia

```
T:>>java TestMap
{Berta=343, Sabina=420, Kasia=42}
343
T:>
```

Kiedy poprosimy o wyświetlenie zawartości mapy, pojawi się ona w formie par klucz=wartość, zapisanych w nawiasach klamrowych (`{ }`); a nie, jak to było w przypadku list i zbiorów, w nawiasach kwadratowych (`[]`).

W końcu wracamy do typów ogólnych

Pamiętasz zapewne, że we wcześniejszej części rozdziału mówiliśmy o tym, że metody pobierające argumenty typów ogólnych mogą być... zakręcone. I mamy tu na myśli „zakręcenie” w sensie polimorficznym. Jeśli odczuwasz, że atmosfera zaczęła się w tym momencie robić dziwna, nie przejmuj się i czytaj dalej — wyjaśnienie całej historii zajmie nam naprawdę tylko kilka stron.

Zaczniemy od przypomnienia jak, pod względem polimorfizmu, działają argumenty będące *tablicami*. Później przejdziemy do analogicznego zastosowania uogólnionych list.

Oto jak to działa w przypadku użycia zwyczajnych tablic:

```
import java.util.*;

public class TestTypowOgolnych1 {
    public static void main(String[] args) {
        new TestTypowOgolnych1().doDziela();
    }

    public void doDziela() {
        Zwierze[] zwierzeta = {new Pies(), new Kot(), new Pies()};
        Pies[] psy = {new Pies(), new Pies(), new Pies()};
        nakarmZwierzeta(zwierzeta);
        nakarmZwierzeta(psy);
    }

    public void nakarmZwierzeta(Zwierze[] zwierzeta) {
        for(Zwierze z: zwierzeta) {
            z.jedz();
        }
    }
}
```

Deklarujemy i tworzymy tablice obiektów `Zwierze`, zawierającą zarówno psy, jak i koty.

Deklarujemy i tworzymy tablice obiektów `Pies`, zawierającą wyłącznie psy

Wywołujemy metodę `nakarmZwierzeta()`, używając tablic obu typów jako argumentów wywołania...

Kluczowe znaczenie ma metoda `nakarmZwierzeta()`, która akceptuje argumenty typu `Zwierze[]` lub `Pies[]`, ponieważ klasa `Pies` JEST typu `Zwierze`. Polimorfizm w dzia³aniu.

Pamiętaj, ze w tym miejscu mo¿emy wywoływać wyłącznie metody zadeklarowane w klasie `Zwierze`, gdyż parametr `zwierzeta` jest tablicą typu `Zwierze`, a my nie zrobiliśmy żadnego jawnego rzutowania. (No bo do czego mielibyśmy rzutować? Przecież ta tablica mo¿e zawierać zarówno obiekty klasy `Pies`, jak i `Kot`).

```
abstract class Zwierze {
    void jedz() {
        System.out.println("zwierz wcina");
    }
}
```

```
class Pies extends Zwierze {
    void szczekaj() { }
}
```

```
class Kot extends Zwierze {
    void miaucz() { }
}
```

Uproszczona hierarchia klasy `Zwierze`.

Jeśli argumentem metody jest tablica obiektów `Zwierze`, to w jej wywo³aniu będzie tamb;e mo¿na przekazaæ tablicê obiektów klas potomnych klasy `Zwierze`.

Innymi slowami, jeśli metoda zosta³a zadeklarowana jako:

```
void bar(Zwierze[] z) {}
```

To zakładajac, ze klasa `Pies` rozszerza `Zwierze`, bêdzie mo¿liwe u¿yc obu poni¿szych wywo³an:

```
bar(tablicaObiektowZwierze);
bar(tablicaObiektowPies);
```

Stosowanie argumentów polimorficznych i typów ogólnych

A zatem przypomnieliśmy sobie, jak działało przekazywanie tablic jako polimorficznych argumentów w wywołaniach metod. Ale czy przekazywanie argumentów będzie działać tak samo, jeśli zmienimy zwyczajną tablicę na kolekcję `ArrayList`? Brzmi to całkiem sensownie, prawda?

W pierwszej kolejności sprawdzmy, co się stanie, gdy spróbujemy użyć kolekcji `ArrayList` zawierającej obiekty `Zwierze`.

Przekazywanie kolekcji `ArrayList<Zwierze>`

```
Prosta zamiana ze Zwierze[] na ArrayList<Zwierze>.
public void doDziela() {
    ArrayList<Zwierze> zwierzeta = new ArrayList<Zwierze>();
    zwierzeta.add(new Pies());
    zwierzeta.add(new Kot()); ← Musimy dodawać po jednym obiekcie, gdyż nie ma żadnego skróconego zapisu pozwalającego na tworzenie kolekcji z zawartością.
    zwierzeta.add(new Pies());
    nakarmZwierzeta(zwierzeta); ← To ten sam kod, jednak tym razem zmienna „zwierzeta” jest kolekcją ArrayList, a nie tablicą.
}
}

public void nakarmZwierzeta(ArrayList<Zwierze> zwierzeta) {
    for(Zwierze z: zwierzeta) {
        z.jedz();
    }
}
```

Tym razem metoda wymaga przekazania kolekcji `ArrayList`, a nie tablicy; cała reszta pozostaje bez zmian. Pamiętaj, że nowe, rozszerzone pętle `for` mogą operować zarówno na tablicach, jak i na kolekcjach.

Nową wersję naszego testowego programu można skompilować i wykonać bez żadnych problemów:

```
Wiersz polecenia
T:\>java TestTypowOgolnych2
zwierz wcina
zwierz wcina
zwierz wcina
T:\>_
```

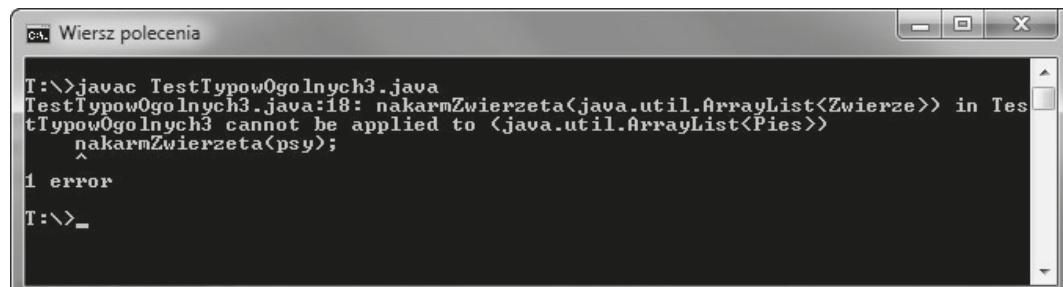
A czy ten kod zadziała, gdy zmienimy kolekcję na ArrayList<Pies>?

Dzięki polimorfizmowi kompilator pozwolił nam przekazać tablice obiektów Pies w wywoaniu metody, której argument został zadeklarowany jako tablica obiektów Zwierze. Z tym nie był problemu. No i oczywiście można przekazać kolekcję `ArrayList<Zwierze>` do metody, której argument zadeklarowano jako `ArrayList<Zwierze>`. Powstaje zatem pytanie, czy w wywoaniu takiej metody będzie można przekazać kolekcję `ArrayList<Pies>`? Skoro takie rozwiązanie zadziałało w przypadku tablic, czemu nie powinno zadziałać także teraz?

Przekazywanie kolekcji `ArrayList<Pies>`

```
public void doDziela() {  
    ArrayList<Zwierze> zwierzeta = new ArrayList<Zwierze>();  
    zwierzeta.add(new Pies());  
    zwierzeta.add(new Kot());  
    zwierzeta.add(new Pies());  
    nakarmZwierzeta(zwierzeta); ← Wiemy, że to wywołanie działa doskonale.  
  
    ArrayList<Pies> psy = new ArrayList<Pies>();  
    psy.add(new Pies());  
    psy.add(new Pies());  
    nakarmZwierzeta(psy); ← Tworzymy kolekcję obiektów Pies  
    i dodajemy kilka obiektów.  
}  
  
Czy to wywołanie zadziała, skoro w wywoaniu  
przekazujemy kolekcję ArrayList obiektów Pies?  
  
public void nakarmZwierzeta(ArrayList<Zwierze> zwierzeta) {  
    for(Zwierze z: zwierzeta) {  
        z.jedz();  
    }  
}
```

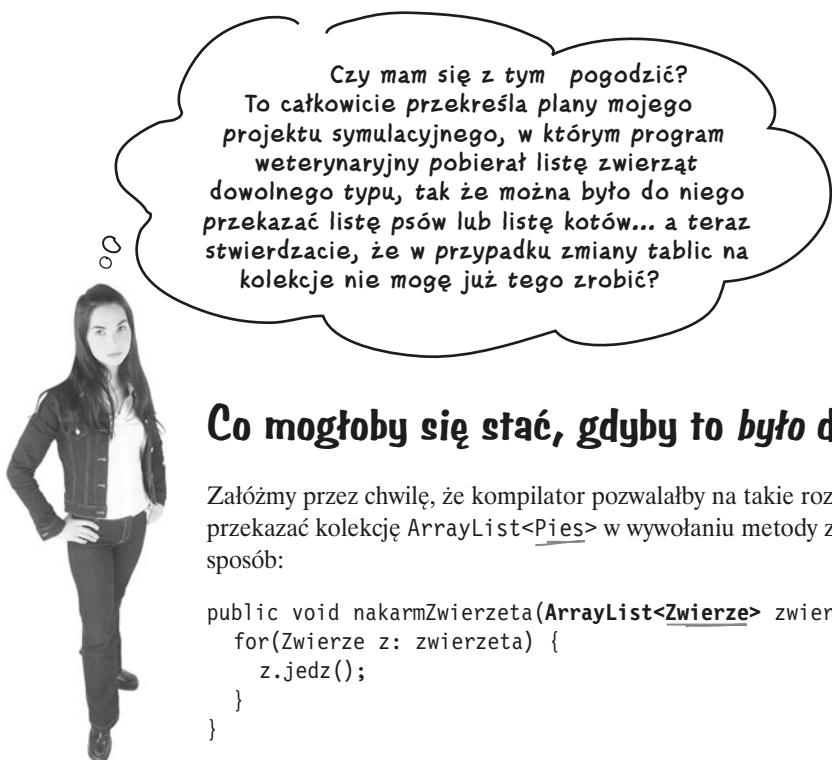
Kiedy spróbujemy skompilować ten kod:



The screenshot shows a terminal window titled "Wiersz polecenia". The command entered is "javac TestTypowOgolnych3.java". The output shows a compilation error at line 18:

```
T:\>javac TestTypowOgolnych3.java  
TestTypowOgolnych3.java:18: nakarmZwierzeta(java.util.ArrayList<Zwierze>) in Tes  
tTypowOgolnych3 cannot be applied to <java.util.ArrayList<Pies>>  
        ^  
        nakarmZwierzeta(psy);  
1 error  
T:\>_
```

Pomyśl wydawał się dobry, a okazał totalną porażkę...



Co mogłoby się stać, gdyby to było dozwolone...

Załóżmy przez chwilę, że kompilator pozwalałby na takie rozwiązania. Zatem pozwoliłby Ci przekazać kolekcję `ArrayList<Pies>` w wywoaniu metody zadeklarowanej w następujący sposób:

```
public void nakarmZwierzeta(ArrayList<Zwierze> zwierzeta) {
    for(Zwierze z: zwierzeta) {
        z.jedz();
    }
}
```

W powyższym kodzie nie ma niczego, co mogłoby się wydawać niebezpieczne, prawda? W końcu cała idea polimorfizmu polega na tym, że wszystko to, co może zrobić obiekt `Zwierze` (w naszym przypadku chodzi o metodę `jedz()`), może także zrobić obiekt `Pies`. A zatem co jest złego w próbie wywołania metody `jedz()` na odwołaniach do obiektów `Pies`?

Nic. Całkowicie nic.

Ten kod jest pod każdym względem w porządku. Wyobraź sobie jednak poniższy kod:

```
public void nakarmZwierzeta(ArrayList<Zwierze> zwierzeta) {
    zwierzeta.add(new Kot());
```

Ups!! Nie możemy dodać obiektu Kot do kolekcji, która być może akceptuje jedynie obiekty Pies.

A zatem na tym polega problem. Bez wątpienia nie ma nic złego w dodawaniu obiektów `Kot` do kolekcji zadeklarowanej jako `ArrayList<Zwierze>`, w końcu właśnie po to tworzy się kolekcje, używając typów bazowych, takich jak `Zwierze` — aby móc w nich umieszczać obiekty dowolnych typów potomnych.

Jeśli jednak przekażemy do metody kolekcję `ArrayList<Pies>` — czyli kolekcję, w której z założenia chcemy przechowywać jedynie obiekty `Pies` — to nagle może się okazać, że w naszej kolekcji znajdzie się obiekt `Kot`. Kompilator wie, że jeśli pozwoli Ci przekazać kolekcję `ArrayList<Pies>` w wywoaniu takiej metody, to ktoś mógłby w trakcie wykonywania programu dodać kota do naszej kolekcji psów. I dlatego kompilator nie pozwoli Ci podjąć takiego ryzyka.

Jeśli zadeklarujesz kolekcję jako `ArrayList<Zwierze>`, to kompilator pozwoli Ci przekazać do niej WYŁĄCZNIE obiekt `ArrayList<Zwierze>`, a nie `ArrayList<Pies>` lub `ArrayList<Kot>`.

Tablice kontra kolekcje ArrayList



Chwileczkę... Jeśli faktycznie powodem naszych problemów z przekazaniem obiektu `ArrayList<Pies>` w wywołaniu metody zadeklarowanej jako `ArrayList<Zwierze>` była chęć uniemożliwienia nam potencjalnego umieszczenia obiektu Kot w kolekcji akceptującej wyłącznie obiekty Pies, to dlaczego analogiczne rozwiązanie działało w przypadku zwykłych tablic? Czy w ich przypadku nie będziemy mieli tego samego problemu? Czy w analogicznej sytuacji nie możemy dodać obiektu Kot do tablicy `Pies[]`?

W przypadku tablic typy są ponownie sprawdzane podczas wykonywania programu... jednak w przypadku kolekcji kontrola jest wykonywana jedynie podczas komilacji.

Załóżmy, że uda Ci się dodać obiekt Kot do tablicy zadeklarowanej jako `Pies[]` (i przekazanej w wywołaniu metody, której argument zadeklarowano jako `Zwierze[]`; co w przypadku tablic, jest rozwiązaniem całkowicie poprawnym i dozwolonym).

```
public void dodziała() {  
    Pies[] psy = {new Pies(), new Pies(), new Pies()};  
    nakarmZwierzeta(psy);  
}
```

```
public void nakarmZwierzeta(Zwierze[] zwierzeta) {  
    zwierzeta[0] = new Kot();  
}
```

Umieściliśmy nowy obiekt Kot w tablicy obiektów Pies. Kompilator pozwolił na to, gdyż wie, że do metody można przekazać zarówno tablicę obiektów Kot, jak i Zwierze. A zatem z punktu widzenia kompilatora istniała możliwość, by taka operacja była prawidłowa.

Ups! Przynajmniej wirtualna maszyna Javy zdecydowała się wkrącić do akcji.

```
T:\>java TestTypowOgolnych1  
Exception in thread "main" java.lang.ArrayStoreException: Kot  
        at TestTypowOgolnych1.nakarmZwierzeta(TestTypowOgolnych1.java:30)  
        at TestTypowOgolnych1.dodziała(TestTypowOgolnych1.java:26)  
        at TestTypowOgolnych1.main(TestTypowOgolnych1.java:5)  
T:\>_
```

Czyż nie byłoby cudownie,
gdyby istniał jakiś sposób na stosowanie
polimorficznych typów kolekcji jako argumentów
metod... tak by mój program weterynaryjny mógł
akceptować zarówno kolekcje obiektów Pies, jak
i Kot? W ten sposób mogłabym wywołać metodę
szczepienie() dla wszystkich obiektów w kolekcji. I żeby
jednocześnie rozwiązanie wciąż było bezpieczne
i uniemożliwiało dodanie obiektu Kot do listy
operującej jedynie na obiektach Pies. Ale wiem,
że to są jedynie moje fantazje...



Znaki wieloznaczne śpieszą z pomocą

To dziwne, lecz okazuje się, że *istnieje* sposób pozwalający na stworzenie takiego argumentu metody, który pozwala przekazywać do niej kolekcje dowolnego typu potomnego klasy `Zwierze`. Najprostszym rozwiązaniem jest wykorzystanie **znaku wieloznacznego** — nowej możliwości języka, dodanej do Javy właśnie w tym celu.

```
public void nakarmZwierzeta(ArrayList<? extends Zwierze> zwierzeta) {  
    for (Zwierze z: zwierzeta) {  
        z.jedz();  
    }  
}
```

Na pewno zastanawiasz się teraz: „Na czym więc polega *różnica*? Czy nie mamy teraz tego samego problemu, co wcześniej? Powyższa metoda nie wykonuje żadnych niebezpiecznych operacji — wywołuje metodę, która muszą mieć wszystkie klasy potomne klasy `Zwierze`. Czy jednak ktoś nie może zmienić jej kodu i dodać obiekt `Kot` do kolekcji `zwierząt`, która w rzeczywistości będzie kolekcją `ArrayList<Pies>?` A ponieważ operacje na kolekcjach nie są sprawdzane podczas wykonywania programu, zatem czym ten zapis różni się od poprzedniego, w którym nie używaliśmy znaku wieloznacznego?”

Twoje rozważania byłyby słuszne. Lecz odpowiedź na Twoje wątpliwości brzmi — NIE. W przypadku zastosowania w deklaracji znaku wieloznacznego `<?>` kompilator nie zezwoli na dodawanie elementów do kolekcji!

Pamiętasz... w tym kontekście słowo kluczowe „`extends`” oznacza zarówno „implementować”, JAK I „rozszerzać”. Zależnie od użytego typu. A zatem, jeśli chcemy przekazywać kolekcje `ArrayList` typów implementujących interfejs `ZwierzątkDomowy`, możemy zadeklarować argument jako:

`ArrayList<? extends ZwierzątkDomowy>`

Jeśli w deklaracji argumentu metody zastosujesz znak wieloznaczny, kompilator NIE POZWOLI Ci na wykonywanie jakichkolwiek czynności, które mogłyby zaszkodzić kolekcji reprezentowanej przez parametr metody.

Wciąż możesz wywoływać metody obiektów przechowywanych w kolekcji, jednak dodawanie elementów do kolekcji nie jest możliwe.

Innymi słowy, możesz operować na elementach kolekcji, lecz nie możesz dodawać do niej nowych elementów. Dzięki temu jesteś zabezpieczony przed błędami występującymi w trakcie wykonywania programu, a to dlatego, że kompilator nie pozwoli na wykonanie jakiejkolwiek operacji, która mógłaby do takich błędów doprowadzić.

A zatem tę operację możesz wykonać wewnątrz metody `nakarmZwierzeta()`:

```
for (Zwierze z: zwierzeta) {  
    z.jedz();  
}
```

Lecz takiego kodu nie udałoby się skompilować:
`zwierzeta.add(new Kot());`

Alternatywna składnia zapewniająca te same możliwości

Zapewne pamiętasz postać deklaracji metody `sort()`, którą zobaczyliśmy w jej dokumentacji... Był w niej używany typ ogólny, jednak zapisano go w niezwykły sposób — parametr typu został umieszczony przed typem wartości wynikowej. Jest to jedynie inny sposób deklarowania parametru typu, efekty jego zastosowania są takie same jak przy użyciu znaku wieloznacznego:

Ta deklaracja:

```
public <T extends Zwierze> void nakarmZwierzeta(ArrayList<T> zwierzeta)
```

Ma to samo znaczenie, co poniższa:

```
public void nakarmZwierzeta(ArrayList<? extends Zwierze> zwierzeta)
```

Nie istnieja
grupie pytania

P: Jeśli obie powyższe deklaracje mają to samo znaczenie, to którą z nich należy wybrać?

O: Wszystko zależy do tego, czy parametrów typu „T” będziesz chciał użyć jeszcze w jakimś innym miejscu kodu. Na przykład, co zrobić, gdybyśmy chcieli, by metoda pobierała dwa argumenty — dwie listy obiektów, których typ rozszerza klasę `Zwierze`? W takim przypadku wygodniejszym rozwiązaniem będzie jednokrotne zadeklarowanie parametru typu:

```
public <T extends Zwierze> void nakarmZwierzeta(ArrayList<T> l1, ArrayList<T> l2)
```

niż dwukrotne używanie znaków wieloznacznych:

```
public void nakarmZwierzeta(ArrayList<? extends Zwierze> l1,  
                           ArrayList<? extends Zwierze> l2)
```



BĄDŹ kompilatorem, ćwiczenie zaawansowane



Twoim zadaniem jest odegranie roli kompilatora i określenie, które z poniższych instrukcji uda się skompilować. Jednak niektóre z nich nie były analizowane w tym rozdziale, dlatego będziesz musiał samemu określić prawidłową odpowiedź, oceniając sytuację na podstawie zamieszczonych wcześniej „reguł”. W niektórych przypadkach będziesz musiał zgadywać, chodzi jednak o to, byś podał sensowną odpowiedź na podstawie zdobytych już wiadomości.

(Uwaga: założ, że przedstawione poniżej instrukcje są umieszczone wewnętrz prawidłowo napisanej klasy i metody).

Czy to można skompilować?

- `ArrayList<Pies> psy1 = new ArrayList<Zwierze>();`
- `ArrayList<Zwierze> zwierzeta1 = new ArrayList<Pies>();`
- `List<Zwierze> lista = new ArrayList<Zwierze>();`
- `ArrayList<Pies> psy = new ArrayList<Pies>();`
- `ArrayList<Zwierze> zwierzeta = psy;`
- `List<Pies> listaPsow = psy;`
- `ArrayList<Object> obiekty = new ArrayList<Object>();`
- `List<Object> objList = obiekty;`
- `ArrayList<Object> objs = new ArrayList<Pies>();`

```

import java.util.*;
public class SortowanieGor {
    LinkedList<Gora> mtn = new LinkedList<Gora>();

    class NazwaCompare implements Comparator<Gora> {
        public int compare(Gora g1, Gora g2) {
            return g1.nazwa.compareTo(g2.nazwa);
        }
    }

    class WysokoscCompare implements Comparator<Gora> {
        public int compare(Gora g1, Gora g2) {
            return (g2.wysokosc - g1.wysokosc); ←
        }
    }

    public static void main(String [] args) {
        new SortowanieGor().doDziela();
    }

    public void doDziela() {
        mtn.add(new Gora("Kasprowy", 1987));
        mtn.add(new Gora("Koscielec", 2155));
        mtn.add(new Gora("Swinica", 2301));
        mtn.add(new Gora("Rysy", 2499));
        System.out.println("Bez sortowania: \n" + mtn);
        NazwaCompare nc = new NazwaCompare();
        Collections.sort(mnt, nc);
        System.out.println("Wg. nazwy:\n" + mtn);
        WysokoscCompare wc = new WysokoscCompare();
        Collections.sort(mnt, wc);
        System.out.println("Wg. wysokosci:\n" + mtn);
    }
}

class Gora {
    String nazwa;
    int wysokosc;

    Gora(String n, int w) {
        nazwa = n;
        wysokosc = w;
    }

    public String toString() {
        return nazwa + " " + wysokosc;
    }
}

```

Rozwiązanie ćwiczenia „Inżynieria wstępna”.

Czy zauważysz, że lista wysokości jest posortowana MALEJĄCO?

Wyniki:

```

C:\ Wiersz poleceń
T:>java SortowanieGor
Bez sortowania:
[Kasprowy 1987, Koscielec 2155, Swinica 2301, Rysy 2499]
Wg. nazwy:
[Kasprowy 1987, Koscielec 2155, Rysy 2499, Swinica 2301]
Wg. wysokosci:
[Rysy 2499, Swinica 2301, Koscielec 2155, Kasprowy 1987]
T:>

```

Rozwiążanie ćwiczenia

Możliwe odpowiedzi:

Comparator,

Comparable,

compareTo(),

compare(),

tak,

nie.

Dysponując następującą instrukcją:

```
Collections.sort(mojaArrayList);
```

Odpowiedz na pytanie:

1. Jaki interfejs musi implementować klasa, której obiekty są przechowywane w kolekcji ArrayList? Comparable
2. Jaką metodę musi implementować klasa, której obiekty są przechowywane w kolekcji ArrayList? compareTo()
3. Czy klasa obiektów przechowywanych w kolekcji ArrayList może jednocześnie implementować oba interfejsy — Comparator **oraz** Comparable? tak

Dysponując następującą instrukcją:

```
Collections.sort(mojaArrayList, mojKomparator);
```

Odpowiedz na pytanie:

4. Czy klasa obiektów przechowywanych w kolekcji ArrayList może implementować interfejs Comparable? tak
5. Czy klasa obiektów przechowywanych w kolekcji ArrayList może implementować interfejs Comparator? tak
6. Czy klasa obiektów przechowywanych w kolekcji ArrayList musi implementować interfejs Comparable? nie
7. Czy klasa obiektów przechowywanych w kolekcji ArrayList musi implementować interfejs Comparator? nie
8. Jaki interfejs musi implementować klasa obiektu przekazanego jako mojKomparator? Comparator
9. Jaką metodę musi implementować klasa obiektu przekazanego jako mojKomparator? compare()

BĄDŹ kompilatorem, rozwiązanie ćwiczenia



Czy to można skompilować?

- `ArrayList<Pies> psy1 = new ArrayList<Zwierze>();`
- `ArrayList<Zwierze> zwierzeta1 = new ArrayList<Pies>();`
- `List<Zwierze> lista = new ArrayList<Zwierze>();`
- `ArrayList<Pies> psy = new ArrayList<Pies>();`
- `ArrayList<Zwierze> zwierzeta = psy;`
- `List<Pies> listaPsow = psy;`
- `ArrayList<Object> obiekty = new ArrayList<Object>();`
- `List<Object> objList = obiekty;`
- `ArrayList<Object> objs = new ArrayList<Pies>();`

17. Pakiety, archiwa JAR i wdrażanie

Rozpowszechnij swój kod

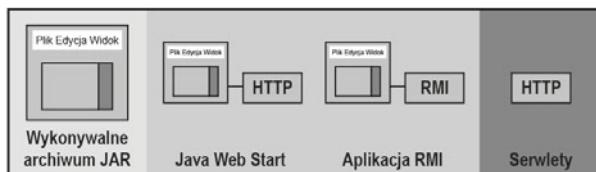


Czas wyruszyć w świat. Napisałś swój program. Przetestowałeś go. Dopracowałeś i ulepszyłeś. Powiedziałś wszystkim znajomym, że wcale byś się nie zmartwił, gdybyś już nigdy w życiu nie napisał ani jednego wiersza kodu. Ale w końcu stworzyłeś dzieło sztuki. Twój program naprawdę działa! Ale co teraz? *Jak* go dostarczysz użytkownikom? I w zasadzie *co* im dostarczysz? Co zrobić, jeśli nawet nie będziesz wiedzieć, kim są użytkownicy Twojego programu? W ostatnich dwóch rozdziałach niniejszej książki przedstawimy zagadnienia związane z organizacją, pakowaniem oraz wdrażaniem kodu napisanego w Javie. Opiszymy możliwości lokalnego, półlokalnego i zdalnego wdrażania oprogramowania, prezentując: pliki wykonywalne JAR, technologię Java Web Start, RMI oraz serwlety. W tym rozdziale skoncentrujemy się głównie na zagadnieniach organizacji i pakowania kodu, które będziesz musiał znać niezależnie od ostatecznego sposobu jego wdrażania. W ostatnim rozdziale zajmiemy się jednymi z najciekawszych i najbardziej ekscytujących rzeczy, jakie można robić w Javie. Odpuszcz się. Wdrażanie kodu nie jest ostatecznym pożegnaniem. Zawsze będziesz go musiał aktualizować i utrzymywać.

Wdrażanie aplikacji

W zasadzie czym jest aplikacja napisana w Javie? Innymi słowy, kiedy już zakończysz etap jej tworzenia, to co masz rozpowszechniać? Istnieje duże prawdopodobieństwo, że system komputerowy użytkownika Twojej aplikacji nie będzie taki sam jak Twój. Najważniejsze jest jednak to, że użytkownicy nie mają Twojej aplikacji. Nadszedł zatem czas, aby nadać Twójemu dziełu postać, w której będzie je można udostępnić „całemu światu”. W tym rozdziale przyjrzymy się możliwościom wdrażania lokalnego, w tym także wykonywalnym archiwom JAR oraz technologii Java Web Start umożliwiającej wdrażanie częściowo lokalne i częściowo zdalne. W następnym rozdziale zajmiemy się możliwościami wdrażania zdalnego, takimi jak RMI oraz serwlety.

Możliwości wdrażania



całkowicie lokalne

mieszanego

całkowicie zdalne

1 Wdrażanie lokalne.

Cała aplikacja działa na komputerze użytkownika lokalnego jako niezależny program prawdopodobnie wyposażony w graficzny interfejs użytkownika i uruchamiany przy wykorzystaniu wykonywalnego archiwum JAR.
(Archiwami JAR zajmiemy się w dalszej części rozdziału).

2 Kombinacja wdrażania lokalnego i zdalnego.

Rozpowszechniana jest „kliencka” część aplikacji, która działa na lokalnym komputerze użytkownika i łączy się z serwerem, gdzie działają pozostałe części aplikacji.

3 Wdrażanie zdalne.

Cała aplikacja działa na serwerze, a klient korzysta z niej za pośrednictwem rozwiązań, które nie muszą mieć nic wspólnego z Java, na przykład, za pośrednictwem przeglądarki WWW.

Zanim zajmiemy się właściwymi zagadnieniami związanymi z wdrażaniem aplikacji, cofniemy się nieco i zobaczymy, co się dzieje, gdy zakończymy etap tworzenia kodu i zgromadzimy pliki klasowe, które następnie zostaną udostępnione użytkownikom końcowym. Co tak naprawdę znajduje się w katalogu roboczym?

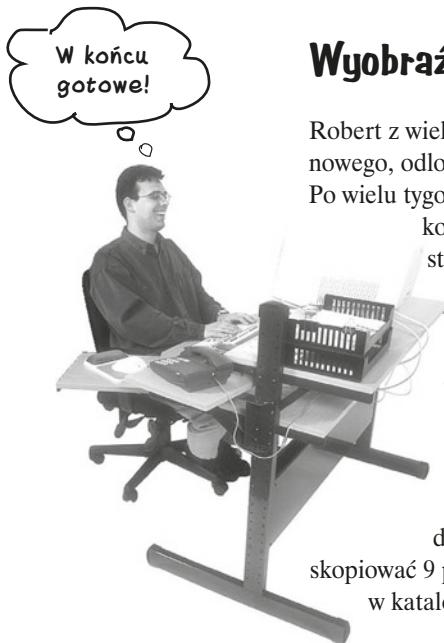
Program napisany w Javie to grupa klas. To właśnie klasy powstają w wyniku tworzenia programu.

Prawdziwe pytanie brzmi:
Co należy zrobić z tymi klasami, kiedy program zostanie już stworzony?



Jakie są zalety i wady rozpowszechniania aplikacji jako programu lokalnego wykonywanego w całości na komputerze użytkownika?

Jakie są zalety i wady rozpowszechniania programu napisanego w Javie w formie systemu działającego na serwerze, a którego kod jest tworzony w formie serwletów i z którego użytkownicy korzystają za pośrednictwem przeglądarek WWW?



Wyobraź sobie taką sytuację...

Robert z wielką satysfakcją pracuje nad ostatnią częścią swojego nowego, odlotowego programu. Napisanego oczywiście w Javie.

Po wielu tygodniach pracy, będąc na etapie „brakuje mi jeszcze jednej kompilacji”, Robert naprawdę skończył. Program jest stosunkowo złożoną i wyszukaną aplikacją o graficznym interfejsie użytkownika, jednak przeważająca większość aplikacji to kod obsługujący interfejs użytkownika i komponenty Swing, zatem Robert stworzył jedynie 9 własnych klas.

W końcu nadszedł czas, aby dostarczyć program klientowi. Ponieważ na komputerze użytkownika już jest zainstalowany Java API, zatem Robert doszedł do wniosku, że w celu dostarczenia aplikacji wystarczy skopiować 9 plików klasowych. Zaczął od wykonania polecenia `dir` w katalogu, w którym przechowywał wszystkie kody źródłowe...



O rany! Stało się coś dziwnego. Zamiast 18 plików (9 plików źródłowych i 9 skompilowanych plików klasowych) Robert zobaczył 31 plików, z których kilka miało bardzo dziwne nazwy, takie jak:

`Konto$PlikListener.class`

`Wykres$ZapiszListener.class`

i tak dalej. Robert całkowicie zapomniał, że kompilator musi wygenerować pliki klasowe dla wszystkich klas wewnętrznych, jakie stworzył dla odbiorców zdarzeń obsługujących interfejs użytkownika. To właśnie ich pliki klasowe mają takie dziwne nazwy.

Teraz Robert musi uważnie wybrać wszystkie niezbędne pliki klasowe. Jeśli ominie choćby jeden, jego program nie będzie działać. Jednak nie jest to proste zadanie, gdyż Robert nie chce wysłać do klienta żadnego z plików źródłowych, a wszystkie pliki są umieszczone w jednym katalogu, tworząc w nim straszny bałagan.

Oddzielanie kodu źródłowego od plików klasowych

Umieszczenie kodów źródłowych oraz plików klasowych w jednym katalogu powoduje straszny bałagan. Jak się okazuje, Robert powinien od samego początku zorganizować swoje pliki, umieszczając kody źródłowe i pliki klasowe w osobnych katalogach. Innymi słowy, powinien się upewnić, że skompilowane pliki klasowe nie będą zapisywane w tym samym katalogu co kod źródłowy programu.

W tym przypadku, kluczowe znaczenie ma połączenie odpowiedniej struktury katalogów oraz opcji -d kompilatora.

Istnieją dziesiątki sposobów organizowania plików, a Twoja firma może wymagać jakiegoś szczególnego rozwiązania. My jednak polecamy rozwiązanie, które stało się niemal standardem.

Rozwiązanie to polega na stworzeniu katalogu projektu, a wewnątrz niego dwóch kolejnych katalogów — **zrodła** oraz **pliki klasowe**. Na początku, podczas pisania programu, kody źródłowe umieszczane są w katalogu **zrodła**. Cała sztuczka polega na tym, aby wyniki kompilacji (czyli pliki *.class*) trafiły do katalogu **pliki klasowe**.

Istnieje bardzo miła opcja kompilatora — **-d** — która pozwala to zrobić.

Kompilacja przy wykorzystaniu opcji -d

```
>cd MojProjekt/zrodla
>javac -d ..//pliki klasowe MojaAplikacja.java
```

Ta opcja informuje kompilator, że skompilowany kod (pliki klasowe) należy umieścić „w katalogu pliki klasowe, do którego z bieżącego katalogu można dotrzeć, wychodząc do katalogu na wyższym poziomie i ponownie schodząc o jeden poziom w dół”.

Także w tym przypadku ostatnim parametrem jest nazwa pliku źródłowego, który należy skompilować.

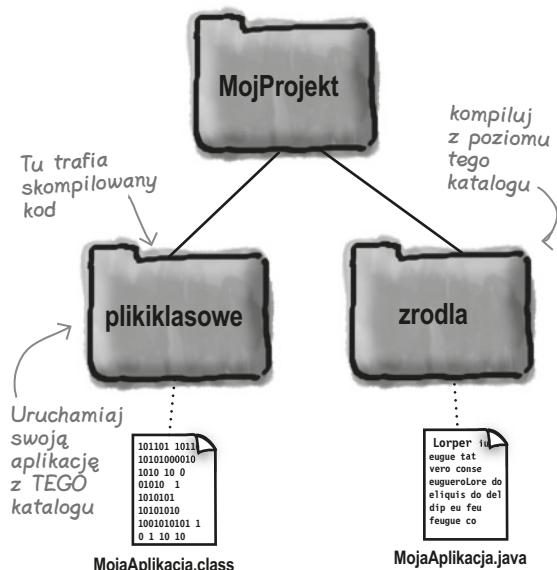
Stosując opcję **-d**, to Ty musisz określić, w jakim **katalogu** zostaną umieszczone skompilowane pliki klasowe, i nie musisz godzić się na rozwiązanie domyślne, polegające na umieszczaniu plików klasowych w tym samym katalogu, w jakim znajdują się pliki źródłowe. Użyj poniższego polecenia, aby skompilować wszystkie pliki *.java* umieszczone w katalogu źródłowym:

```
>javac -d ..//pliki klasowe *.java
```

Uruchamianie kodu

```
>cd MojProjekt/pliki klasowe
>java Aplikacja
```

Używając wyrażenia **.java*, można skompilować **WSZYSTKIE** pliki źródłowe znajdujące się w bieżącym katalogu.
Uruchamianie aplikacji z katalogu zawierającego pliki źródłowe.



(porada: wszystkie rozważania i przykłady przedstawione w tym rozdziale zakładają, że bieżący katalog roboczy, czyli „..”, należy do ścieżki CLASSPATH. Jeśli sam określesz wartość zmiennej środowiskowej CLASSPATH, upewnij się, że zawiera ona ciąg „..”)

Umieszczanie programów w archiwach JAR



Plik **JAR** to archiwum Java (ang. Java ARchive). Bazuje on na formacie plików *pkzip* i pozwala na umieszczenie wszystkich klas w jednym pliku, dzięki czemu nasz klient otrzyma nie 28 plików klasowych, lecz jedno archiwum **JAR**. Jeśli znasz polecenie tar stosowane w systemach UNIX, to zapewne opcje programu jar wydadzą Ci się znajome. (Uwaga: kiedy słowo *JAR* piszemy wielkimi literami, mamy na myśli *archiwum JAR*; natomiast to samo słowo pisane małymi literami — jar — oznacza *program narzędziowy* służący do tworzenia plików *JAR*).

Należy sobie zadać pytanie, co nasz klient może zrobić z plikiem *JAR*? W jaki sposób może go *uruchomić*?

Wystarczy stworzyć *wykonywalny* plik *JAR*.

Stworzenie wykonywalnego pliku *JAR* oznacza, że przed wykonaniem programu użytkownik końcowy nie będzie musiał rozpakowywać archiwum i zapisywać wszystkich plików klasowych gdzieś poza nim. Użytkownik będzie mógł uruchomić aplikację pomimo to, że pliki klasowe wciąż będą umieszczone w pliku *JAR*. Cała sztuka polega na stworzeniu pliku **manifestu**, który zostaje umieszczony wewnątrz archiwum i zawiera informacje o znajdujących się w nim plikach. Aby stworzyć wykonywalny plik *JAR*, *manifest musi określić, jaka klasa zawiera metodę main()*!

Tworzenie wykonywalnego pliku JAR

- Upewnij się, że **wszystkie pliki klasowe znajdują się w katalogu plikiklasowe**.

W dalszej części rozdziału usprawnimy to rozwiązanie, jednak na razie przyjmij, że **wszystkie pliki klasowe znajdują się w jednym katalogu plikiklasowe**.

- Stwórz plik **manifest.txt** określający, która klasa zawiera metodę **main()**.

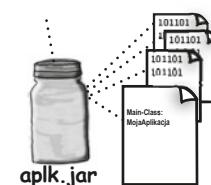
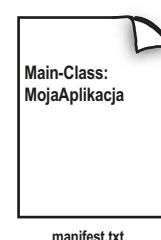
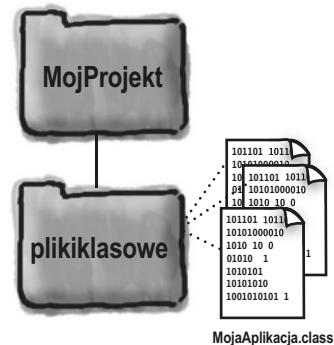
Stwórz plik tekstowy o nazwie **manifest.txt** zawierający następujący wiersz:

Main-Class: MojaAplikacja ← Nie dodawaj końcówki .class

Po wpisaniu całej zawartości tego wiersza wcisnij klawisz Enter, gdyż w przeciwnym razie plik manifestu może nie działać poprawnie. Umieść plik manifestu w katalogu **plikiklasowe**.

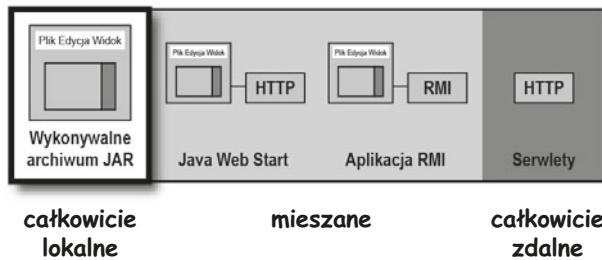
- Przy użyciu programu narzędziowego jar stwórz archiwum **JAR** zawierające **wszystkie pliki znajdujące się w katalogu plikiklasowe** — czyli **wszystkie pliki klasowe, jak również plik manifestu**.

```
>cd MojProjekt/plikiklasowe
>jar -cvmf manifest.txt apk.jar *.class
lub:
>jar -cvmf manifest.txt apk.jar MojaAplikacja.class
```



W archiwum JAR nie ma żadnego kodu źródłowego (plików z rozszerzeniami .java).

Wykonywalne archiwum JAR



Lokalne aplikacje Javy są rozpowszechniane i wdrażane w formie wykonywalnych archiwów JAR.

Uruchamianie (wykonywanie) archiwum JAR

Java (a właściwie wirtualna maszyna Javy) jest w stanie wczytywać klasę z archiwum JAR i wykonać jej metodę `main()`. W rzeczywistości cała aplikacja może być umieszczona w pliku JAR. Kiedy cała zabawa się już rozpoczęcie (czyli gdy zostanie wywołana metoda `main()`), wirtualna maszyna Javy nie zwraca uwagi na to, skąd pochodzą pliki klasowe, o ile tylko jest w stanie je odnaleźć. A jednym z miejsc, w jakich JVM poszukuje plików klasowych, są archiwia JAR umieszczone w ścieżce dostępu do klas. Jeśli tylko wirtualna maszyna Javy odnajdzie archiwum JAR, przejrzyszy jego zawartość, jeśli będzie musiała odszukać i wczytać klasę.



Wirtualna maszyna Javy musi „zobaczyć” plik JAR, a zatem musi się on znajdować w jednym z katalogów wskazanych w zmiennej środowiskowej CLASSPATH. Najprostszym sposobem zapewnienia, że plik JAR będzie „widoczny”, jest umieszczenie go w aktualnym katalogu roboczym.

JVM szuka w archiwum pliku manifestu, a w nim wiersza Main-Class. Jeśli nie uda się znaleźć pliku bądź niezbędnego wiersza tekstu, zostanie zgłoszony wyjątek.

W zależności od sposobu skonfigurowania systemu operacyjnego możesz być nawet w stanie uruchomić plik *JAR*, klikając go dwukrotnie. Możliwość ta jest dostępna w większości wersji systemu operacyjnego Windows oraz w systemie Mac OS X. W większości przypadków ten sam efekt można osiągnąć, zaznaczając plik *JAR* i nakazując systemowi operacyjnemu, aby „Otworzył go za pomocą...” (lub innego równoważnego polecenia dostępnego w używanym systemie operacyjnym).

Nie, istnieją głupie pytania

P: Dlaczego nie można umieścić w archiwum JAR całego katalogu?

O: Wirtualna maszyna Javy zagląda do archiwum *JAR* i oczekuje, że znajdzie to, czego szuka, bez żadnych dodatkowych kłopotów. Nie będzie poszukiwać klasy w żadnych innych katalogach, chyba że klasa ta należy do jakiegoś pakietu. Jednak nawet gdy poszukiwana klasa należy do pakietu, to JVM będzie jej szukać wyłącznie w katalogach odpowiadających instrukcji importu.

P: Przepraszam, co powiedzieliście?

O: Owszem — nie możesz umieścić plików klasowych w jakimkolwiek dowolnym katalogu i wraz z nim dodać ich do archiwum *JAR*. Jeśli jednak klasa należy do pakietu, to możesz umieścić w pliku *JAR* całą strukturę katalogów tego pakietu. W rzeczywistości nawet *musisz* tak zrobić. Wyjaśnimy to wszystko na następnej stronie, więc się nie denerwuj.

Umieść klasy w pakietach

A zatem stworzyłeś swoje wspaniałe klasy nadające się do wielokrotnego wykorzystania, skompilowałeś je i umieściłeś w firmowej bibliotece, aby inni programiści mogli ich używać. Pławiąc się w chwale i dumie wynikającej z oddania najlepszych przykładów programowania obiektowego, jakie kiedykolwiek stworzono (oczywiście to była tylko Twoja skromna opinia), odbierasz telefon. Ktoś z wściekłością wykrzykuje coś do słuchawki. Okazuje się, że dwie spośród oddanych przez Ciebie klas, mają takie same nazwy jak klasy, które ledwie chwilę temu Franek dodał do biblioteki. I to jest przyczyną katastrofy, gdyż kolizje nazw i niejednoznaczności uniemożliwiają pracę całego zespołu.

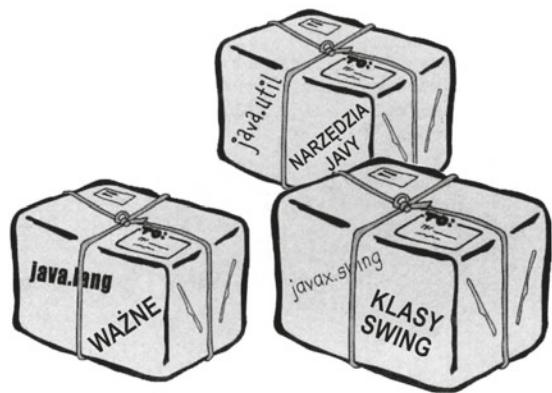
A wszystko to dla tego, że nie umieściłeś swoich klas w pakietach! No... w zasadzie używałeś pakietów, w sensie wykorzystywania klas należących do Java API, które oczywiście są w pakietach. Jednak własnych klas nie umieściłeś w pakietach, a w praktyce jest to karygodne zaniedbanie.

Dlatego też zmodyfikujemy nieco strukturę organizacji kodu przedstawioną we wcześniejszej części rozdziału — bardzo nieznacznie: ograniczymy się do umieszczenia naszych klas w pakiecie i zapisaniu całego pakietu w archiwum *JAR*. Powinieneś zwrócić baczną uwagę na subtelne i drobne szczegóły. Nawet najdrobniejsza pomyłka może uniemożliwić skompilowanie bądź uruchomienie kodu.

Pakiety zapobiegają konfliktom nazw klas

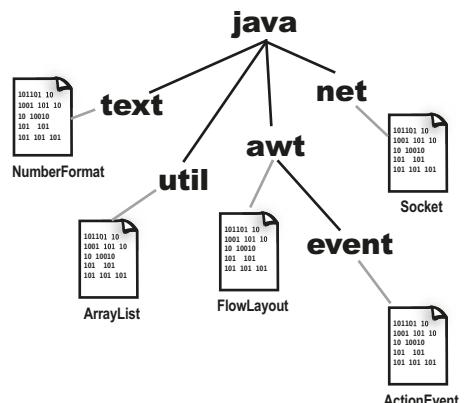
Choć pakiety nie służą wyłącznie zapobieganiu konfliktom nazw, to jednak właśnie to jest podstawową przyczyną ich stosowania. Możesz stworzyć klasy Klient, Konto oraz Koszyk. Ale wiesz co? Połowa programistów tworzących korporacyjne aplikacje do handlu elektronicznego prawdopodobnie stworzyła klasy o takich samych nazwach. W świecie programowania zorientowanego obiekto to bardzo niebezpieczna sytuacja. Programiści muszą mieć możliwość łączenia komponentów pochodzących z wielu różnych źródeł i wykorzystania ich w celu stworzenia czegoś nowego. Twoje komponenty muszą być w stanie „współpracować z innymi” komponentami, nawet tymi, których nie stworzyłeś sam, lub których istnienia nawet nie podejrzewasz.

Pamiętasz, w rozdziale 6. wspominaliśmy, że nazwa pakietu to jak gdyby pełna nazwa klasy, technicznie rzecz biorąc, jest ona określana jako nazwa w pełni kwalifikowana. Klasa *ArrayList*, to tak naprawdę klasa *java.util.ArrayList*, JButton to *javax.swing.JButton*, a Socket — *java.net.Socket*. Zwróć uwagę, że w przypadku klas *ArrayList* oraz *Socket*, pierwszą częścią ich w pełni kwalifikowanych nazw jest „java”. Myśląc o strukturze pakietów, myśl o hierarchii i na jej podstawie organizuj swoje klasy.



Struktura pakietów dla klas należących do Java API:

`java.text.NumberFormat`
`java.util.ArrayList`
`java.awtFlowLayout`
`java.awt.event.ActionEvent`
`java.net.Socket`



Co Ci przypomina ten rysunek?
 Czy nie wygląda on podobnie do hierarchii katalogów?

Określanie nazw pakietów

...no i w końcu zdecydowałam się nadać mojej klasie kwantowej symulacji piekarnika nazwę `tmp.tst.Heisenberg`.



Och! Dlaczego?! Tę samą nazwę chciałam nadać mojej klasie reprezentującej subatomowe żelazko! Chyba będę musiała wymyślić jakąś inną nazwę.

Pakietы могут запирать конфликты имен, однако для этого необходимо иметь уверенность, что имена пакетов будут уникальными.

Наиболее простым и лучшим решением является предварительное назначение имен пакетов именам доменов, записанным в обратном порядке.

com.headfirstjava.Ksiazka
↓
nazwa pakietu ↑
 nazwa klasy

Zapobieganie konfliktom nazw pakietów

Umieszczanie klas w pakietach zmniejsza szansę występowania konfliktów nazw pomiędzy samymi klasami. Czy jednak można w jakiś sposób uniemożliwić programistom stosowanie tych samych nazw pakietów? Innymi słowy, czy można zapobiec sytuacji, w której dwóch programistów, z których każdy dysponuje klasą `KontoKlienta`, chce umieścić tę klasę w pakiecie o nazwie `zakupy.klient`? W takim przypadku obie klasy wciąż miałyby tę samą nazwę:

`zakupy.klient.KontoKlienta`

Firma Sun zaleca stosowanie konwencji nazewniczej, która w znacznym stopniu ogranicza prawdopodobieństwo wystąpienia takiej sytuacji — polega ona na poprzedzeniu nazwy klasy nazwą domeny zapisaną w odwrotnej kolejności. Pamiętaj, że nazwy domen z założenia są unikalne. Dwie osoby mogą mieć te same imiona i nazwiska — Bartolomeusz Brzdegiel — jednak dwie domeny nie mogą mieć tej samej nazwy `barb.com.pl`

Odwrocone nazwy domen w nazwach pakietów

→ **com.headfirstjava.projekty.Wykres** ← Nazwa klasy zawsze zaczyna się wielką literą.

Zacznij nazwę pakietu od nazwy domeny zapisanej w odwrotnej kolejności, rozdziel poszczególne części tej nazwy kropkami, a następnie dodaj do niej nazwę odpowiadającą wykorzystanej strukturze organizacji klasy.

↑
projekty.Wykres może być często stosowaną nazwą, jednak dodanie `com.headfirstjava` oznacza, że konflikty mogą wystąpić tylko w obrębie naszego projektu.

Aby umieścić klasy w pakiecie:

1 Wybierz nazwę pakietu.

Do prezentowanych przykładów wybraliśmy nazwę `com.headfirstjava`. Nazwa klasy to `PrzykladPakietu`, a zatem w pełni kwalifikowaną nazwą klasy będzie: `com.headfirstjava.PrzykladPakietu`.

2 W pliku źródłowym klasy umieść instrukcję package.

Musi to być pierwsza instrukcja w pliku, umieszczona nawet przed instrukcjami importu. W jednym pliku źródłowym można umieścić tylko jedną instrukcję package, co sprawia, że wszystkie klasy zdefiniowane w jednym pliku źródłowym będą należeć do tego samego pakietu. Oczywiście dotyczy to także klas wewnętrznych.

```
package com.headfirstjava;

import javax.swing.*;

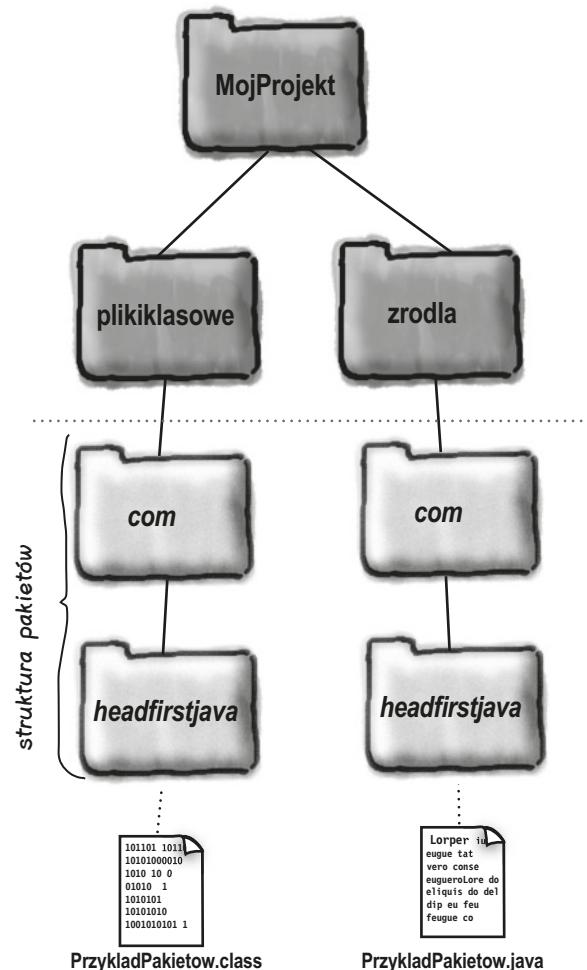
public class PrzykladPakietu {
    // kod, który odmieni Twoje życie
}
```

3 Utwórz strukturę katalogów odpowiadającą nazwom pakietów.

Nie wystarczy stwierdzić, że klasa należy do pakietu, umieszczając odpowiednią instrukcję w jej kodzie źródłowym. Klasa nie będzie tak naprawdę należeć do pakietu, jeśli nie zostanie umieszczona w odpowiedniej strukturze katalogów. A zatem, jeśli w pełni kwalifikowaną nazwą klasy jest `com.headfirstjava.PrzykladPakietow`, to musisz umieścić plik źródłowy tej klasy w katalogu `headfirstjava`, który z kolei musi znajdować się w katalogu `com`.

Choć można skompilować klasę bez umieszczania jej w odpowiedniej strukturze katalogów, to jednak nie jest to warte problemów, jakie mogą przy tym wystąpić. Zatem przechowuj pliki źródłowe w strukturze katalogów, która odpowiada strukturze pakietów, a unikniesz masy problemów i niepotrzebnego bólu głowy.

Musisz umieścić klasy w strukturze katalogów, która odpowiada hierarchii pakietów.



Zarówno w katalogu służącym do przechowywania plików źródłowych, jak i plików klasowych stwórz strukturę katalogów odpowiadającą hierarchii pakietów.

Skompiluj i uruchom program, używając pakietów

Kompilacja i uruchamianie programu, w którym wykorzystywane są pakiety

Jeśli klasa należy do pakietu, to jej skompilowanie i wykonanie jest nieco bardziej złożone. Podstawowy problem polega na tym, że zarówno kompilator, jak i wirtualna maszyna Java muszą znaleźć samą klasę, jak i wszystkie pozostałe klasy, których ona używa. W przypadku klas należących do Java API nie jest to problemem.

Java zawsze wie, gdzie szukać własnych klas. Z kolei w przypadku klas tworzonych przez Ciebie, rozwiązanie polegające na skompilowaniu klas z poziomu katalogu, w którym się one znajdują, nie zda egzaminu (a przynajmniej nie będzie działać *niezawodnie*). Gwarantujemy jednak, że sobie poradzisz, jeśli postąpisz w sposób opisany na tej stronie. Oczywiście istnieją także inne sposoby, lecz uważaemy, że ten jest najpewniejszy i najbliższyszy.

Kompilacja przy użyciu flagi -d (directory)

> cd MojProjekt/zrosla ← Pozostań w katalogu źródła — NIE przechodź do katalogów zawierających plik źródłowy (z rozszerzeniem .java)!

> javac -d ../plikiKlasowe com/headfirstjava/PrzykladPakietu.java

Informuje kompilator, że skompilowany kod (pliki klasowe) należy umieścić w tym katalogu, w strukturze katalogów odpowiadającej strukturze pakietów!!! Tak, tak, kompilator wie o tym.

Teraz musisz podać ŚCIEŻKĘ dostępu do pliku zawierającego kod źródłowy klasy.

Poniższe polecenie pozwala skompilować wszystkie pliki źródłowe należące do pakietu com.headfirstjava:

> javac -d ../plikiZrodlowe com/headfirstjava/*.java

Ten znak pozwoli skompilować wszystkie pliki źródłowe (.java) umieszczone we wskazanym katalogu.

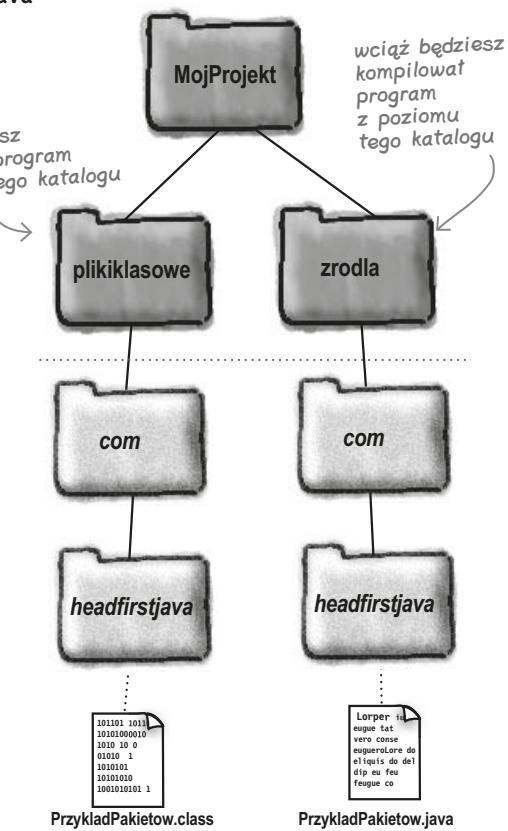
Uruchamianie programu

> cd MojProjekt/plikiKlasowe

Uruchom swój program z poziomu katalogu plikiKlasowe.

> java com.headfirstjava.PrzykladPakietu

MUSISZ podać w pełni kwalifikowaną nazwę klasy! Wirtualna maszyna Java zauważa nazwę klasy i zajrzy do bieżącego katalogu, oczekując, że znajdzie w nim katalog o nazwie com, w nim katalog o nazwie headfirstjava, a dopiero w nim uruchamiany plik klasowy. Jeśli plik klasowy znajdzie się w katalogu com lub nawet w katalogu plikiKlasowe, to programu nie da się uruchomić!



Flaga -d jest nawet lepsza, niż twierdziliśmy

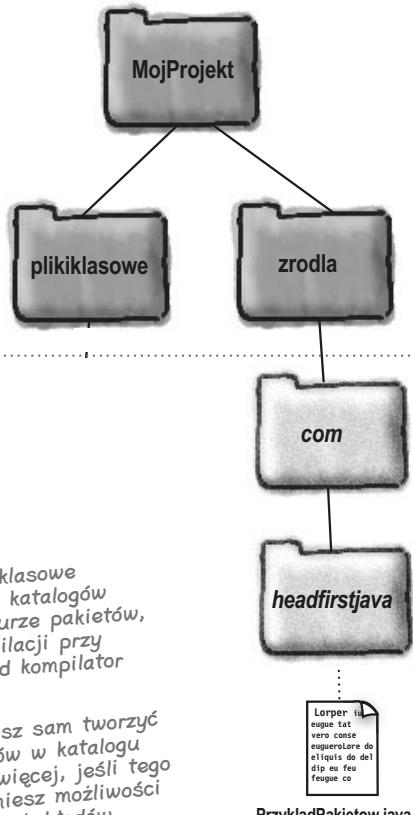
Kompilacja przy wykorzystaniu flagi -d jest wspaniałą, gdyż nie tylko pozwala na umieszczenie plików klasowych w innym katalogu niż ten, w którym znajdują się pliki źródłowe, lecz także wie, że należy je umieścić w odpowiedniej strukturze katalogów odpowiadającej strukturze pakietów.

Ale to nie wszystko!

Załóżmy, że stworzyłeś śliczną strukturę katalogów przygotowaną do rozmieszczenia kodu źródłowego Twojego programu. Ale nie stworzyłeś odpowiedniej struktury w katalogu, w którym mają być umieszczane pliki klasowe. To żaden problem! Użycie opcji -d podczas komplikacji nie tylko nakazuje umieszczenie plików klasowych w odpowiednim drzewie katalogów, lecz także pozwala na *stworzenie* niezbędnych katalogów, jeśli te nie będą istnieć.

Jeśli w katalogu plikiKlasowe nie istnieje struktura katalogów odpowiadająca strukturze pakietów, to w przypadku komplikacji przy wykorzystaniu flagi -d kompilator ją utworzy.

A zatem nie musisz sam tworzyć struktury katalogów w katalogu plikiKlasowe. Co więcej, jeśli tego nie zrobisz, unikniesz możliwości popełniania prostych błędów typograficznych.



Flaga -d instruuje kompilator: „Umieść klasy w strukturze katalogów odpowiadającej strukturze pakietu, używając jako katalogu głównego katalogu podanego po flagie -d. Ale... jeśli niezbędnych katalogów tam nie będzie, to najpierw je utwórz, a dopiero potem umieszczaj klasy w odpowiednich miejscach.”

Nie istnieją
główne pytania

P: Przeszedłem do katalogu, w którym była moja klasa, a wirtualna maszyna Javy oznajmia mi teraz, że nie może jej znaleźć! A przecież ona tam jest — bezpośrednio w bieżącym katalogu!

O: Kiedy klasa należy do pakietu, to nie możesz jej uruchamiać, posługując się jej „krótką” nazwą. W wierszu poleceń musisz podać w pełni kwalifikowaną nazwę klasy, której metodę `main()` chcesz uruchomić. Jednak w pełni kwalifikowana nazwa klasy określa strukturę *pakietu* i dlatego Java wymaga, aby klasa znajdowała się w odpowiedniej strukturze katalogów. A zatem, jeśli w wierszu poleceń wpiszesz:

% java com.tmp.Ksiazka

to wirtualna maszyna Javy spróbuje znaleźć w bieżącym katalogu (oraz we wszystkich innych katalogach podanych w zmiennej środowiskowej CLASSPATH) katalog o nazwie *com*. JVM nie zacznie szukać klasy *Ksiazka*, jeśli wcześniej nie znajdzie katalogu *com*, w którym znajduje się katalog *tmp*. Tylko pod tym warunkiem wirtualna maszyna Javy uzna, że znalazła właściwą klasę *Ksiazka*. Jeśli klasa zostanie znaleziona w jakimkolwiek innym miejscu, JVM uzna, że nie należy ona do odpowiedniej struktury, i to nawet w sytuacji, jeśli w rzeczywistości należy! Na przykład JVM nie zajrzy do katalogu nadzrzednego i nie stwierdzi: „Ocho, widzę, że mamy tutaj katalog o nazwie *com*, a zatem to musi być odpowiedni pakiet”.

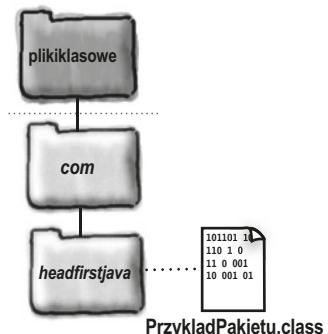
Tworzenie wykonywalnego archiwum JAR zawierającego pakiety



Jeśli klasa należy do pakietu, to cała struktura katalogów musi się znaleźć w pliku JAR! Nie można tak po prostu umieszczać klas w archiwum w taki sam sposób, w jaki robiliśmy to przed zastosowaniem pakietów. Należy także zwrócić uwagę, aby nie umieszczać w archiwum żadnych katalogów znajdujących się „ponad” pakietem. Innymi słowy, pierwszy katalog pakietu musi być pierwszym katalogiem umieszczonym w pliku JAR! Gdybyś przez przypadek umieścił w archiwum jakiś katalog znajdujący się *ponad* pakietem (w naszym przykładzie mógłby to być katalog *plikiklasowe*) to archiwum nie działałoby poprawnie.

Tworzenie wykonywalnego archiwum JAR

- Upewnij się, że wszystkie pliki klasowe znajdują się w poprawnej strukturze katalogów odpowiadającej strukturze pakietu i umieszczonej w katalogu plikiklasowe.

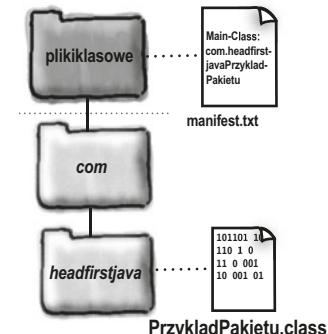


- Stwórz plik manifest.txt określający klasę, która ma metodę main() i nie zapomnij podać w pełni kwalifikowanej nazwy tej klasy!

Stwórz zwyczajny plik tekstowy o nazwie manifest.txt i umieść w nim jeden wiersz tekstu:

Main-Class: com.headfirstjava.PrzykladPakietu

Plik manifest.txt umieść w katalogu plikiklasowe.

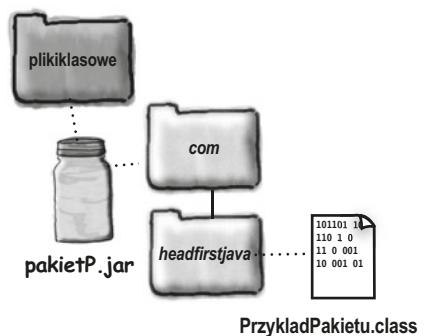


- Używając programu narzędziowego jar, stwórz archiwum JAR zawierające katalogi pakietu oraz plik manifestu.

Podczas tworzenia pliku JAR należy wskazać wyłącznie katalog com, a w archiwum znajdzie się cały pakiet (i wszystkie klasy należące do niego).

```
> cd MojProjekt/plikiklasowe  
> jar -cvmf manifest.txt pakietP.jar com
```

Wystarczy wskazać jedynie katalog com! Wraz z nim w archiwum znajdzie się wszystko, czego potrzebujesz!



Gdzie jest umieszczany plik manifestu?

A może by tak zziejrzeć do archiwum *JAR* i to sprawdzić? Możliwości programu narzędziowego jar nie ograniczają się do tworzenia archiwów *JAR*. Można także pobierać zawartość archiwów (tak samo jak w przypadku zwykłych archiwów *ZIP* lub *TAR*).

Umieszczamy plik JAR w katalogu o nazwie *testaplk*

Wyobraź sobie, że umieściłeś plik *pakietP.jar* w katalogu *testaplk*.

Polecenia programu jar służące do tworzenia listy zawartości oraz rozpakowywania archiwum:

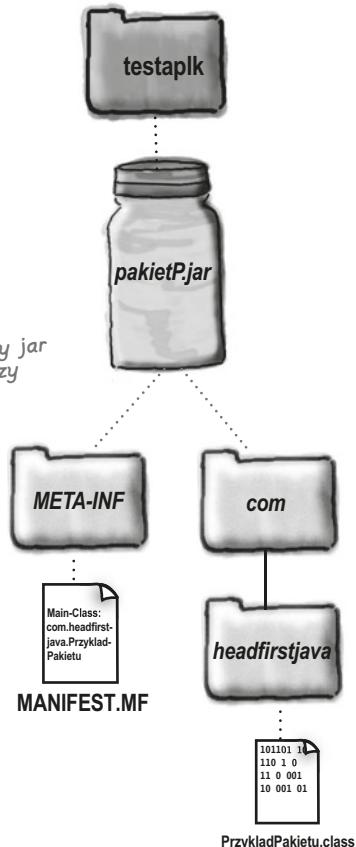
① Sporządzenie listy zawartości archiwum JAR

> jar -tf pakietP.jar

Litery „tf” pochodzą od angielskich słów „table file”, co można przetłumaczyć jako „stwórz listę zawartości pliku”.

```
Wiersz polecenia
c:\ Wiersz polecenia
T:\>cd testaplk
T:\testaplk>jar -tf pakietP.jar
META-INF/
META-INF/MANIFEST.MF
com/
com/headfirstjava/
com/headfirstjava/PrzykladPakietu.java
T:\testaplk>
```

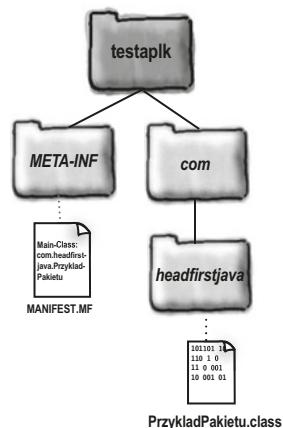
Program narzędziowy jar automatycznie tworzy katalog *META-INF* i umieszcza w nim plik manifestu.



② Pobieranie zawartości archiwum JAR.

> cd testaplk
> jar -xf pakietP.jar

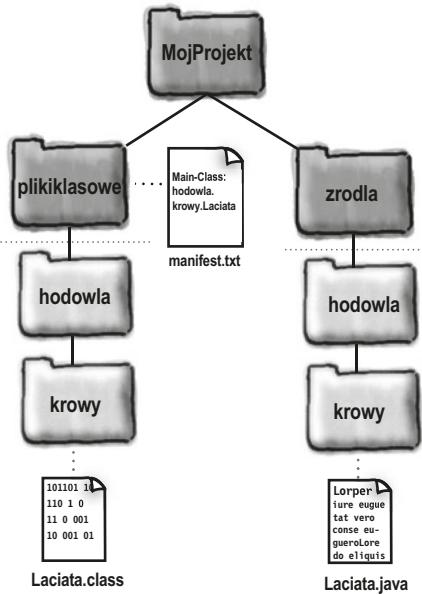
Litery „xf” pochodzą od angielskich słów „extract file”, czyli „pobierz zawartość pliku”. Użycie tej flagi przypomina „rozpakowanie” normalnego archiwum, „rozpakowanie” normalnego archiwum, takiego jak plik ZIP lub TAR. Po takiego jak plik ZIP lub TAR. Po takim rozpakowaniu pliku pakietP.jar w bieżącym katalogu pojawią się dwa nowe katalogi – *META-INF* oraz *com*.



META-INF to skrócona wersja słowa „metainformacje”. Program jar tworzy zarówno katalog *META-INF*, jak i plik *MANIFEST.MF*. Dodatkowo program odczytuje zawartość pliku manifestu i zapisuje ją w pliku *MANIFEST.MF*. Jak widać, Twój plik manifestu nie jest umieszczany w archiwum *JAR*, jednak jego zawartość trafia do „prawdziwego” pliku manifestu (*MANIFEST.MF*).



Zaostrz ołówek



Na podstawie struktury katalogów i pakietów przedstawionej na rysunku, określ, jakie polecenia należy wydać, aby skompilować klasę, wykonać ją, stworzyć wykonywalne archiwum JAR oraz je wykonać. Załóż, że korzystamy ze standardowego rozwiązania, w którym struktura katalogów pakietu rozpoczyna się bezpośrednio poniżej katalogów *zrodła* oraz *plikiklasowe*. Innymi słowy, katalogi *zrodła* oraz *plikiklasowe* nie stanowią części pakietu.

Kompilacja:

> cd zrodla
> javac _____

Uruchomienie:

> cd _____
> java _____

Stworzenie archiwum JAR:

> cd _____
> _____

Uruchomienie klasy z archiwum JAR:

> cd _____
> _____

Pytanie dodatkowe: Co jest nie tak z nazwą pakietu?

Nie istniejąca grupa pytań

P: Co się stanie, kiedy spróbuję uruchomić wykonywalny plik JAR, a użytkownik końcowy nie będzie miał zainstalowanej Javy?

O: Nic się nie stanie, gdyż bez wirtualnej maszyny Javy nie można uruchomić żadnego programu napisanego w Javie.

P: W jaki sposób można zainstalować Javę na komputerze użytkownika końcowego?

O: W optymalnym przypadku możesz stworzyć własny program instalacyjny i rozpowszechniać go razem z aplikacją. Kilka firm udostępnia programy instalacyjne, a można wśród nich znaleźć zarówno narzędzia stosunkowo proste, jak i programy o ogromnych możliwościach. Na przykład program instalacyjny może wykryć, czy użytkownik ma zainstalowaną odpowiednią wersję Javy, a jeśli nie ma, to będzie w stanie ją zainstalować i skonfigurować przed zainstalowaniem Twojej aplikacji. Firmy InstallShield, InstallAnywhere oraz DeployDirect udostępniają narzędzia przystosowane do instalowania aplikacji napisanych w Javie.

Kolejną wielką zaletą programów instalacyjnych jest to, że korzystając z nich, można nawet stworzyć instalacyjny dysk CD-ROM zawierający programy instalacyjne przeznaczone dla wszystkich głównych platform Javy, a zatem... jeden CD, by nad wszystkim panować. Na przykład, jeśli użytkownik używa systemu operacyjnego Solaris, to zainstalowana zostanie Java w wersji przeznaczonej dla tego systemu. Jeśli użytkownik ma system Windows, to zostanie zainstalowana wersja Javy przeznaczona do użycia w Windows i tak dalej. Jeśli dysponujesz wystarczająco dużym budżetem, to z punktu widzenia użytkownika końcowego jest to bez wątpienia najprostszy sposób zainstalowania i skonfigurowania odpowiedniej wersji Javy.



CELNE SPOSTRZEŻENIA

- Zorganizuj swój projekt w taki sposób, aby kody źródłowe oraz pliki klasowe nie znajdowały się w tym samym katalogu.
- Standardowy sposób organizacji polega na stworzeniu katalogu *projekt*, a w nim katalogów *zrodła* oraz *plikiklasowe*.
- Umieszczanie klas w pakietach zapobiega występowaniu konfliktów nazw, zwłaszcza jeśli przed nazwą klasy zostanie dodana nazwa domeny zapisana w odwrotnej kolejności.
- Aby dodać klasę do pakietu, umieść na samym początku pliku źródłowego instrukcję *package*, powinna się ona znaleźć nawet przed instrukcjami importu:
package com.bardzosprytny;
- Aby klasa należała do pakietu, musi zostać umieszczona w *strukturze katalogów*, która *idealnie odpowiada strukturze pakietu*. W przypadku klasy *com.bardzosprytny.Tmp*, klasa *Tmp* musi się znaleźć w katalogu *bardzosprytny* znajdującym się w katalogu *com*.
- Jeśli chcesz, aby skompilowane pliki klasowe zostały zapisane w odpowiedniej strukturze katalogów umieszczonej w katalogu *plikiklasowe*, to podczas ich komplikacji użyj flagi *-d*:

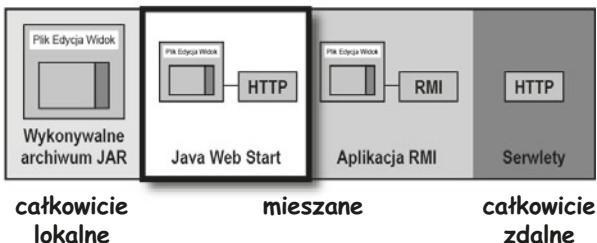
```
> cd zrodla
> javac -d ../plikiklasowe com/bardzosprytny/Tmp.java
```
- Aby uruchomić skompilowany kod, przejdź do katalogu *plikiklasowe* i podaj w pełni kwalifikowaną nazwę klasy:

```
> cd plikiklasowe
> java com.bardzosprytny.Tmp
```
- Możesz zebrać wszystkie klasy i zapisać je w archiwum *JAR* (ang. Java ARchive). Archiwa JAR bazują na formacie *pkzip*.
- Można także stworzyć wykonywalny plik *JAR*. W tym celu do archiwum należy dodać plik manifestu określający klasę, która posiada metodę *main()*. Aby stworzyć plik manifestu, stwórz zwykły plik tekstowy i umieść w nim wiersz tekstu podobny do:
Main-Class: com.bardzosprytny.Tmp
- Nie zapomnij nacisnąć klawisza *Enter* na końcu wiersza umieszczonego w pliku manifestu; jeśli tego nie zrobisz, manifest może nie działać poprawnie.
- Aby stworzyć archiwum *JAR*, użyj polecenia:
jar -cvfm manifest.txt MojJar.jar com
- Cała struktura katalogów pakietu (i tylko ona — bez żadnych dodatkowych katalogów) musi się znaleźć na „głównym poziomie” archiwum *JAR*.
- Aby wykonać kod umieszczony w pliku *JAR*, użyj polecenia:
java -jar MojJar.jar

Czyż nie byłoby cudownie...

Wykonywalne archiwa JAR są świetne, ale czyż nie byłoby cudownie, gdyby istniał jakiś sposób pozwalający tworzyć niezależne programy wyposażone w graficzny interfejs użytkownika, które można by rozpowszechniać przy użyciu internetu? Tak żeby nie trzeba tworzyć i rozpowszechniać tych wszystkich CD-ROM-ów. I czy nie byłoby wspaniale, gdyby program mógł się automatycznie aktualizować, zastępując tylko te swoje fragmenty, które zostały zmienione? Użytkownik zawsze miałby aktualną wersję programu, a Ty nigdy nie musiałbyś się przejmować rozpowszechnianiem jego nowych wersji...





Java Web Start

Dzięki technologii Java Web Start aplikacja może być po raz pierwszy uruchamiana za pośrednictwem przeglądarki WWW (Rozumiesz, skąd pochodzi nazwa technologii?), jednak działa jako zupełnie niezależny program (no... *niemal* niezależny), który nie podlega ograniczeniom narzuconym przez przeglądarkę. A kiedy program zostanie pobrany na komputer użytkownika końcowego (co następuje, gdy użytkownik po raz pierwszy użyje łącza uruchamiającego ten program), już na nim *pozostanie*.

Java Web Start jest (między innymi) niewielkim programem dostępnym na komputerze użytkownika, który działa w podobny sposób jak wiele innych programów dodatkowych wykorzystywanych w przeglądarkach WWW (na przykład podobnie do programu Adobe Acrobat Reader, który jest otwierany, gdy przeglądarka pobierze plik z rozszerzeniem *.pdf*). Program ten określany jest jako **aplikacja pomocnicza Java Web Start**, a jego podstawowym przeznaczeniem jest zarządzanie pobieraniem, aktualizowaniem i wykonywaniem *Twoich* aplikacji JWS.

Kiedy JWS pobiera aplikację (wykonywalny plik *JAR*), uruchamia odpowiednią metodę *main()*. Później użytkownik końcowy będzie już w stanie uruchomić aplikację bezpośrednio z poziomu aplikacji pomocniczej JWS, bez konieczności korzystania z łącza użytego za pierwszym razem.

Ale nie to jest najważniejsze. Zadziwiającą cechą technologii JWS jest jej zdolność do wykrywania nawet drobnych zmian (na przykład jednego pliku klasowego) wprowadzonych w „oryginalnej” aplikacji w serwerze oraz do przeprowadzania aktualizacji kodu bez żadnej ingerencji ze strony użytkownika końcowego.

Oczywiście wciąż pozostaje problem, w jaki sposób użytkownicy mają *zdobyć* Javę oraz technologię Java Web Start? A potrzebują ich obu — Javy, aby uruchomić aplikację, oraz technologię Java Web Start (która sama w sobie została stworzona jako niewielki program Javy), aby obsługiwać pobieranie i uruchamianie aplikacji. Ale rozwiązano nawet *ten* problem. Otóż aplikację można skonfigurować w taki sposób, aby w przypadku, gdy JWS nie jest zainstalowana na komputerze użytkownika końcowego, niezbędne oprogramowanie zostało pobrane z witryny firmy Oracle. Z kolei, jeśli technologia JWS jest zainstalowana, lecz użytkownik ma przestarzałą wersję Javy (tak, w aplikacji JWS możesz określić, jaka wersja Javy jest wymagana do jej uruchomienia), to pobrane zostanie odpowiednie środowisko Java 2 Standard Edition.

Jednak przede wszystkim technologia JWS jest łatwa w użyciu. Aplikacje JWS można udostępniać tak samo jak wszelkie inne zasoby publikowane na WWW, takie jak zwyczajne dokumenty *HTML* bądź obrazy *JPEG*. Wystarczy, że stworzysz stronę WWW zawierającą odpowiednie łącza, i sprawa załatwiona.

W końcu aplikacja JWS nie jest niczym więcej niż wykonywalnym archiwum JAR, które użytkownik końcowy może pobrać z internetu.

Użytkownik końcowy uruchamia aplikację JWS, klikając odpowiednie łącze umieszczone na stronie WWW. Jednak kiedy aplikacja zostanie już pobrana, jest ona uruchamiania niezależnie od przeglądarki, tak jak każdy inny niezależny program napisany w Javie. W rzeczywistości aplikacja JWS jest jedynie wykonywalnym plikiem JAR rozpowszechnianym za pośrednictwem internetu.

Jak działa technologia Java Web Start?

- Użytkownik kliknie łącze do aplikacji JWS (a konkretnie do pliku .jnlp) umieszczone na stronie WWW.

Oto postać łącza umieszczanego na stronie WWW:

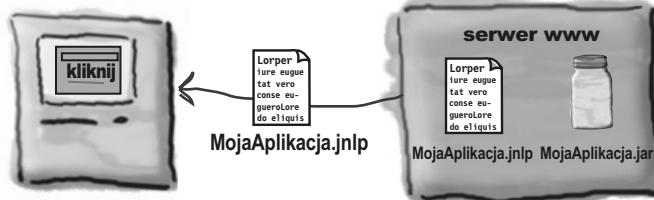
```
<a href="MojaAplikacja.jnlp">Kliknij</a>
```

przeglądarka www



- Serwer WWW odbiera żądanie i przesyła w odpowiedzi plik .jnlp (to NIE jest archiwum JAR).

Plik .jnlp to dokument XML, który określa nazwę wykonywalnego pliku JAR, w którym została zapisana aplikacja.

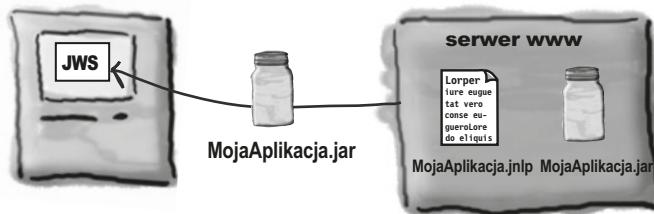


- Przeglądarka WWW uruchamia Java Web Start (niewielką „aplikację pomocniczą” działającą po stronie klienta). Aplikacja ta odczytuje plik .jnlp i prosi serwer o przesłanie pliku MojaAplikacja.jar.

Java Web Start



- Serwer WWW wysyła żądany plik .jar.



- Java Web Start odbiera plik JAR i uruchamia aplikację, wywołując wskazaną metodę main() (identycznie jak dzieje się w przypadku uruchamiania wykonywalnego archiwum JAR).

Następny razem, gdy użytkownik będzie chciał wykonać program, może uruchomić pomocniczą aplikację JWS, a za jej pośrednictwem wybrany program — nawet nie będzie musiał być połączony do internetu.

WebStartWitam (aplikacja w archiwum JAR)



Plik .jnlp

Aby stworzyć aplikację Java Web Start, będziesz potrzebował pliku *.jnlp* (ang. *Java Network Launch Protocol* — protokół uruchamiania przez sieć aplikacji Java) opisującego tę aplikację. To właśnie ten plik odczytuje aplikacja pomocnicza JWS w celu odszukania Twojego archiwum *JAR* i uruchomienia aplikacji (co następuje poprzez wywołanie metody *main()* jednej z klas umieszczonej w tym archiwum). Plik *.jnlp* jest zwyczajnym dokumentem *XML*, w którym można umieszczać wiele różnych informacji; poniżej przedstawiona została absolutnie minimalna, niezbędna postać tego pliku:

```
<?xml version="1.0" encoding="utf-8">
<jnlp spec="0.2.1.0"
      codebase="http://127.0.0.1/~kasia"
      href="MojaAplikacja.jnlp">
```

Atrybut „*codebase*” służy do określenia „katalogu głównego” na serwerze WWW, w którym znajdują się wszystkie pliki związane z Twoją aplikacją. Testujemy tę aplikację lokalnie, zatem podaliśmy adres pętli zwrotnej „127.0.0.1”. Gdyby aplikacja była udostępniana na naszym normalnym serwerze WWW, użylibyśmy adresu „<http://www.wickedlysmart.com>”.

```
<information>
    <title>Aplikacja Kasi</title>
    <vendor>Bardzo Sprytni</vendor>
    <homepage href="index.html" />
    <description>Demonstracja technologii Java Web Start</description>
    <icon href="appkasi.gif" />
    <offline-allowed />
</information>
```

Pamiętaj, aby podać wszystkie te znaczniki, gdyż w przeciwnym razie aplikacja JWS może nie działać poprawnie! Znaczniki te są używane przez pomocniczą aplikację JWS, która używa ich przede wszystkim w celu wyświetlenia stosownych informacji, gdy użytkownik chce ponownie uruchomić pobrany już wcześniej program JWS.

```
<resources>
    <j2se version="1.3+" />
    <jar href="MojaAplikacja.jar" />
</resources>
```

Ten znacznik informuje, że użytkownik może uruchamiać aplikację bez połączenia z internetem. Jeśli użytkownik nie będzie połączony do Sieci, opcja automatycznej aktualizacji zostanie wyłączona.

```
<application-desc main-class="WitajJWS" />
</jnlp>
```

Ten znacznik bardzo przypomina wiersz podawany w pliku manifestu... określa on, która klasa archiwum JAR ma metodę *main()*.

Czynności, jakie należy wykonać w celu stworzenia i wdrożenia aplikacji JWS

- 1 Stwórz wykonywalne archiwum JAR zawierające aplikację.



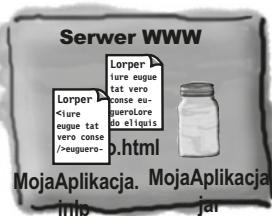
MojaAplikacja.jar

- 2 Napisz plik .jnlp.



MojaAplikacja.jnlp

- 3 Umieść archiwum JAR i plik .jnlp na serwerze WWW.



- 4 Zdefiniuj nowy typ MIME, który będzie obsługiwać serwer WWW.

application/x-java-jnlp-file

Dzięki temu serwer WWW będzie wysyłać plik .jnlp z odpowiednimi nagłówkami, dzięki czemu przeglądarka, odbierając ten plik, będzie wiedzieć, czym on jest, oraz że w celu jego obsługi należy uruchomić pomocniczą aplikację JWS.

Serwer WWW
skonfiguruj typ
mime

- 5 Stwórz stronę WWW zawierającą łącze do pliku .jnlp.

```
<HTML>
  <BODY>
    <a href="MojaAplikacja.jnlp">Uruchom moją aplikację</a>
  </BODY>
</HTML>
```



MojaAppJWS.html



W tym rozdziale przedstawiliśmy zagadnienia związane z pakowaniem, wdrażaniem oraz technologią JWS. Twoim zadaniem jest określenie, czy poniższe stwierdzenia są prawdziwe czy nie.

👉 PRAWDA CZY FAŁSZ 🙌

1. Kompilator Javy udostępnia opcję `-d`, która pozwala określić, gdzie mają być umieszczane pliki klasowe (`.class`).
2. *JAR* to standardowy katalog, w którym są umieszczane pliki klasowe.
3. Tworząc archiwum *JAR*, trzeba stworzyć plik o nazwie *jar.mf*.
4. Dodatkowy plik umieszczany w archiwum *JAR* określa, która klasa zawiera metodę `main()`.
5. Archiwum *JAR* trzeba rozpakować, zanim wirtualna maszyna Javy będzie mogła wykorzystać umieszczone w nich pliki klasowe.
6. Z poziomu wiersza poleceń archiwum *JAR* są uruchamiane przy użyciu flagi `-arch`.
7. Struktura pakietu jest reprezentowana przez hierarchię katalogów.
8. Wykorzystanie nazwy domeny zarezerwowanej dla Twojej firmy nie jest zalecanym sposobem nazywania pakietów.
9. Różne klasy zdefiniowane w jednym pliku źródłowym mogą należeć do różnych pakietów.
10. W przypadku komplikowania klas należących do pakietów zalecane jest stosowanie flagi `-p`.
11. W przypadku komplikowania klas należących do pakietów pełna nazwa klasy musi odzwierciedlać strukturę katalogów.
12. Rozsądne stosowanie flagi `-d` pozwala uniknąć występowania błędów typograficznych w nazwach katalogów.
13. W wyniku rozpakowania archiwum *JAR* zawierającego pakiety utworzony zostanie katalog o nazwie *META-INF*.
14. W wyniku rozpakowania archiwum *JAR* zawierającego pakiety utworzony zostanie plik o nazwie *manifest.mf*.
15. Aplikacja pomocnicza JWS zawsze jest uruchamiana razem z przeglądarką WWW.
16. Aby aplikacja JWS mogła działać poprawnie, potrzebny jest jej plik *.nlp* (ang. *Network Launch Protocol*).
17. Metoda `main()` aplikacji JWS jest określana w archiwum *JAR* tej aplikacji.

Ćwiczenia. Co było pierwsze?



Co było pierwsze

Spójrz na przedstawioną poniżej sekwencję zdarzeń i rozmieśc je w kolejności odpowiadającej etapom uruchamiania aplikacji Java Web Start.



Nie istnieją głupie pytania

P: Czym technologia Java Web Start różni się od appletów?

O: Apletty nie mogą istnieć poza przeglądarką WWW. Apletty stanowią raczej część wyświetlanej strony WWW, niż są pobierane za jej pośrednictwem. Innymi słowy, z punktu widzenia przeglądarki aplet przypomina obrazek JPEG lub jakikolwiek inny zasób. W celu uruchomienia apletu przeglądarka używa bądź to specjalnego dodatku (Java plug-in), bądź środowiska Java wbudowanego bezpośrednio w przeglądarkę (przy czym to rozwiązanie jest aktualnie coraz rzadziej stosowane). Apletty nie dysponują równie dużymi możliwościami funkcjonalnymi (zwłaszcza jeśli chodzi o takie możliwości jak automatyczne aktualizacje), a co więcej, za każdym razem trzeba je pobierać z serwera. W przypadku aplikacji JWS, gdy użytkownik pobierze ją już z serwera, to jej powtórne uruchomienie nie będzie nawet wymagać użycia przeglądarki. Aby uruchomić pobraną wcześniej aplikację JWS, wystarczy posłużyć się aplikacją pomocniczą JWS.

P: Jakie są ograniczenia bezpieczeństwa narzucane na aplikacje Java Web Start?

O: Aplikacje JWS mają pewne ograniczenia, na przykład nie mogą odczytywać ani zapisywać plików na dysku lokalnego komputera. Jednak... Technologia JWS udostępnia swój własny interfejs programistyczny zawierający specjalne wersje okien dialogowych służących do otwierania i zapisywania plików; dzięki temu, dysponując pozwoleniem użytkownika, Twoja aplikacja może zapisywać i odczytywać swoje własne pliki przechowywane w specjalnym, wydzielonym obszarze dysku na komputerze użytkownika.

1.

2.

3.

4.

5.

6.

7.



CELNE SPOSTRZEŻENIA

- Technologia Java Web Start pozwala na wdrażanie niezależnych aplikacji klienckich za pośrednictwem WWW.
- Java Web Start zawiera „aplikację pomocniczą”, która musi być zainstalowana na komputerze użytkownika.
- Aplikacja Java Web Start składa się z dwóch elementów: wykonywalnego pliku `JAR` oraz pliku `.jnlp`.
- Plik `.jnlp` jest zwyczajnym dokumentem `XML` opisującym aplikację JWS. Zawiera on znaczniki określające nazwę aplikacji, położenie pliku `JAR` oraz nazwę klasy zawierającej metodę `main()`.
- Kiedy przeglądarka odbierze plik `.jnlp` (przesłany z serwera w wyniku kliknięcia odpowiednio przygotowanego łącza), uruchamia aplikację pomocniczą JWS.
- Aplikacja pomocnicza JWS odczytuje plik `.jnlp` i prosi serwer WWW o przesłanie wykonywalnego archiwum `JAR`.
- Kiedy aplikacja pomocnicza JWS odbierze archiwum `JAR`, wywołuje metodę `main()` wskazanej klasy (określonej w pliku `.jnlp`).



Rozwiązańa ćwiczeń



1. Użytkownik kliką łącze umieszczone na stronie WWW.
2. Serwer WWW przesyła do przeglądarki plik *.jnlp*.
3. Przeglądarka prosi serwer WWW o przesłanie pliku *.jnlp*.
4. Przeglądarka WWW uruchamia aplikację pomocniczą *JWS*.
5. Aplikacja pomocnicza *JWS* żąda przesłania archiwum *JAR*.
6. Serwer WWW przesyła archiwum *JAR* do aplikacji pomocniczej *JWS*.
7. Aplikacja pomocnicza *JWS* wywołuje metodę *main()* wskazanej klasy umieszczonej w pliku *JAR*.

Prawda

1. Kompilator Javy udostępnia opcję *-d*, która pozwala określić, gdzie mają być umieszczane pliki klasowe (*.class*).

Fałsz

2. *JAR* to standardowy katalog, w którym są umieszczane pliki klasowe.

Fałsz

3. Tworząc archiwa *JAR*, trzeba stworzyć plik o nazwie *jar.mf*.

Prawda

4. Dodatkowy plik umieszczany w archiwum *JAR* określa, która klasa zawiera metodę *main()*.

Fałsz

5. Archiwa *JAR* trzeba rozpakować, zanim wirtualna maszyna Javy będzie mogła wykorzystać umieszczone w nich pliki klasowe.

Fałsz

6. Z poziomu wiersza poleceń archiwa *JAR* są uruchamiane przy użyciu flagi *-arch*.

Prawda

7. Struktura pakietu jest reprezentowana przez hierarchię katalogów.

Fałsz

8. Wykorzystanie nazwy domeny zarezerowanej dla Twojej firmy nie jest zalecany sposobem nazywania pakietów.

Fałsz

9. Różne klasy zdefiniowane w jednym pliku źródłowym mogą należeć do różnych pakietów.

Fałsz

10. W przypadku kompilowania klas należących do pakietów zalecane jest stosowanie flagi *-p*.

Prawda

11. W przypadku kompilowania klas należących do pakietów pełna nazwa klasy musi odzwierciedlać strukturę katalogów.

Prawda

12. Rozsądne stosowanie flagi *-d* pozwala uniknąć występowania błędów typograficznych w nazwach katalogów.

Prawda

13. W wyniku rozpakowania archiwum *JAR* zawierającego pakiety utworzony zostanie katalog o nazwie *META-INF*.

Prawda

14. W wyniku rozpakowania archiwum *JAR* zawierającego pakiety utworzony zostanie plik o nazwie *manifest.mf*.

Fałsz

15. Aplikacja pomocnicza *JWS* zawsze jest uruchamiana razem z przeglądarką WWW.

Fałsz

16. Aby aplikacja *JWS* mogła działać poprawnie, potrzebny jest jej plik *.nlp* (ang. *Network Launch Protocol*).

Fałsz

17. Metoda *main()* aplikacji *JWS* jest określana w archiwum *JAR* tej aplikacji.

18. Zdalne wdrażanie z użyciem RMI

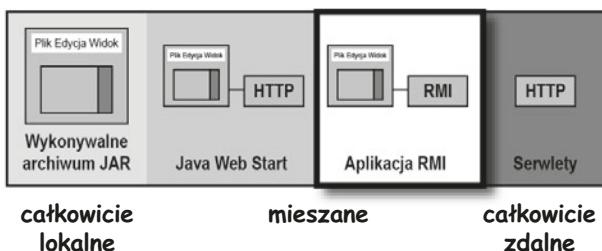
Przetwarzanie rozproszone



Wszyscy twierdzą, że związki na odległość są trudne, ale dzięki RMI stają się łatwe. Niezależnie od tego, jak daleko od siebie jesteśmy, RMI sprawia, że czujemy się, jakbyśmy byli razem.

Odległość nie musi być przeszkodą. Oczywiście, że wszystko jest łatwiejsze, gdy poszczególne części aplikacji znajdują się w jednym miejscu, na jednej sterce i są wykonywane przez jedną wirtualną maszynę Javy. Jednak nie zawsze jest to możliwe, albo pożądane. Na przykład, co zrobić w sytuacji, gdy aplikacja obsługuje bardzo złożone obliczenia, lecz użytkownicy końcowi używają prostych urządzeń wyposażonych w Javę? A co w przypadku, gdy Twoja aplikacja potrzebuje informacji z bazy danych, lecz ze względów bezpieczeństwa dostęp do tej bazy ma wyłącznie kod wykonywany na serwerze? Wyobraź sobie duży serwer obsługujący aplikację do prowadzenia handlu elektronicznego, który musi być obsługiwany przez system transakcyjny? Czasami pewna część aplikacji musi działać na serwerze, a inna jej część — zazwyczaj pełniąca funkcję programu klienta — musi działać na innym komputerze. W tym rozdziale dowiesz się, w jaki sposób można używać zadziwiająco prostej technologii *Remote Method Invocation* (RMI). Oprócz tego побieżnie przyjrzymy się serwletom, komponentom Enterprise JavaBeans (EJB) oraz technologii Jini, jak również sprawdzimy, w jaki sposób EJB oraz technologia Jini są zależne od RMI. Na samym końcu rozdziału — a jednocześnie książki — przedstawimy najbardziej niesamowity program, jaki można napisać w Javie — przeglądarkę uniwersalnych usług.

Ille ster?

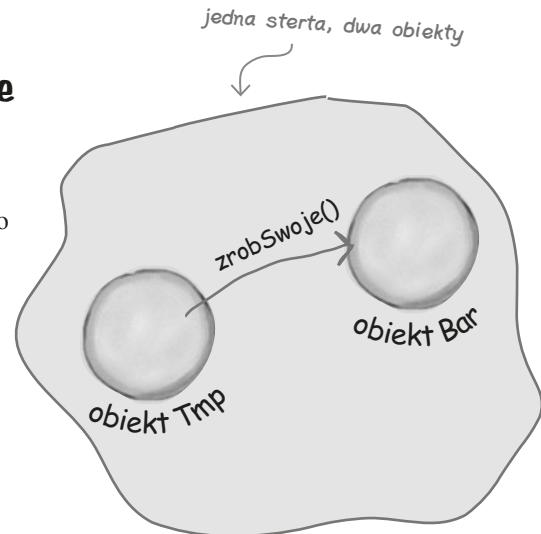


Wywołania metod zawsze są realizowane pomiędzy dwoma obiektami na tej samej stercie

Jak do tej pory obiekt wywołujący metodę oraz obiekt, którego metoda była wywoływana, zawsze były wykonywane przez tę samą, wirtualną maszynę Javy. Innymi słowy, zarówno obiekt wywołujący, jak i wywoływany (czyli ten, którego metoda jest wywoływana) istniały na tej samej stercie.

```
class Tmp {  
    void doRoboty() {  
        Bar b = new Bar();  
        b.zrobSwoje();  
    }  
    public static void main(String[] args) {  
        Tmp t = new Tmp();  
        t.doRoboty();  
    }  
}
```

Analizując powyższy przykład, wiemy, że obiekt `Tmp`, do którego odwołuje się zmienna referencyjna `t` oraz obiekt `Bar`, do którego odwołuje się zmienna referencyjna `b`, istnieją na tej samej stercie i są wykonywane przez tę samą wirtualną maszynę Javy. Pamiętaj, że to właśnie wirtualna maszyna Javy jest odpowiedzialna za zapisanie w zmiennej referencyjnej odpowiednich bitów określających, *jak można się dostać do obiektu przechowywanego na stercie*. Wirtualna maszyna Javy zawsze wie, gdzie znajduje się każdy z obiektów i jak się do niego dostać. Niemniej jednak JVM może znać jedynie odwołania do obiektów znajdujących się na *jej* stercie! Nie jest natomiast możliwe, by wirtualna maszyna Javy działająca na jednym komputerze znała zawartość sterty wirtualnej maszyny Javy działającej na *innym* komputerze. W rzeczywistości JVM działająca na jednym komputerze nie może dysponować żadnymi informacjami na temat innej wirtualnej maszyny Javy działającej na tym samym komputerze. To, czy wirtualne maszyny Javy znajdują się na tych samych czy też na innych fizycznych komputerach, nie ma najmniejszego znaczenia; liczy się wyłącznie to, że obie wirtualne maszyny Javy są... różne.

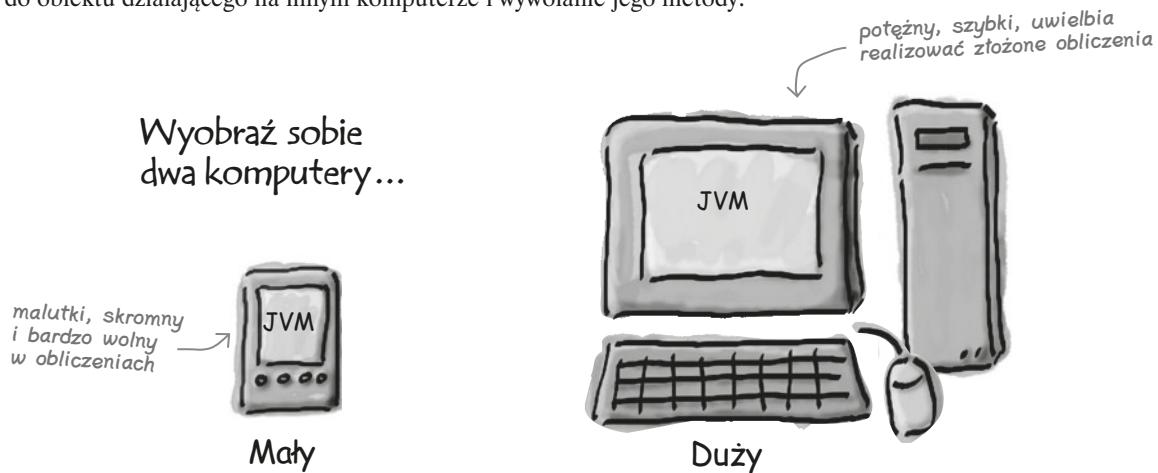


W większości przypadków, kiedy jeden obiekt wywołuje metodę innego obiektu, oba te obiekty znajdują się na tej samej stercie. Innymi słowy, oba obiekty działają w obrębie tej samej wirtualnej maszyny Javy.

A co zrobić, gdy chcemy wywołać metodę obiektu działającego na innym komputerze?

Wiemy, jak przekazać informacje na inny komputer — przy wykorzystaniu gniazd oraz operacji wejścia—wyjścia. Należy utworzyć połączenie, pobrać strumień wyjściowy gniazda i zapisać w nim jakieś dane.

Jednak co zrobić, jeśli będziemy chcieli wywołać metodę czegoś, co działa na innym komputerze... co jest wykonywane przez inną wirtualną maszynę Javy? Oczywiście można by stworzyć własny protokół i po przesłaniu danych na serwer przetwarzać je, określać, co należy zrobić, wykonywać to, co trzeba, i przesyłać wyniki przy wykorzystaniu innego strumienia. Jednak ilu problemów przysporzyłoby takie rozwiązanie... Pomyśl, o ile przyjemniejsze i łatwiejsze byłoby pobranie odwołania do obiektu działającego na innym komputerze i wywołanie jego metody.



Duży ma coś, czego potrzebuje Mały.

Moc obliczeniową.

Mały chciałby coś wysłać do Dużego, żeby Duży wykonał złożone obliczenia.

Mały chciałby jedynie wywołać metodę...

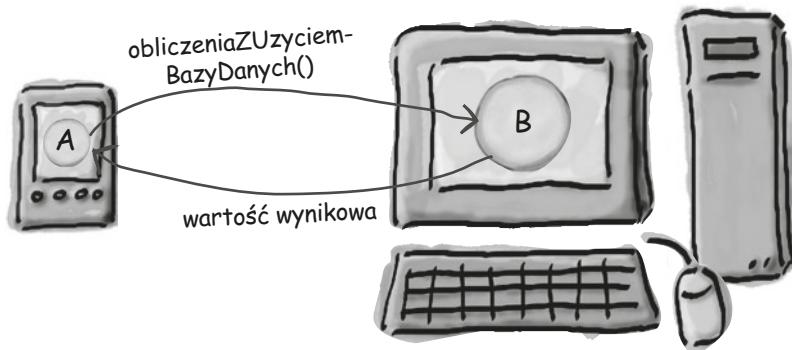
`double obliczeniaZUzyciemBazyDanych(DaneDoObliczen dane)`
i odczytać wyniki.

Ale w jaki sposób Mały może zdobyć odwołanie do obiektu działającego na Dużym?

Dwa obiekty, dwie sterty

Obiekt A działający na komputerze Mały chce wywołać metodę obiektu B działającego na komputerze Duży

Musimy zatem znaleźć odpowiedź na pytanie, w jaki sposób można sprawić, by obiekt działający na jednym komputerze wywołał metodę obiektu działającego na innym komputerze (co oznacza inną stertę oraz inną wirtualną maszynę Javy).



Przecież czegoś takiego nie można zrobić!

Cóż, nie można... a przynajmniej nie bezpośrednio. Nie można pobrać odwołania do obiektu znajdującego się na innej stercie. Jeśli napiszesz:

Pies p = ???

to bez względu na to, do jakiego obiektu będzie się odwoływać zmienna *p*, musi się on znajdować na tej samej stercie co obiekt, do którego należy aktualnie wykonywany kod.

Wyobraź sobie jednak, że chciałbyś zaprojektować rozwiązanie wykorzystujące gniazda i operacje wejścia-wyjścia, które przekaże Twoje intencje (czyli część wywołania metody obiektu działającego na innym komputerze), a jednocześnie sprawi, że cała operacja będzie przypominać wywołanie metody lokalnej.

Innymi słowy, chciałbyś spowodować wywołanie metody *zdalnego* obiektu (czyli obiektu przechowywanego na jakiejś innej stercie), używając w tym celu kodu, który będzie *udawał*, że wywoływana jest metoda obiektu lokalnego. A zatem chciałbyś połączyć prostotę starego, normalnego wywoływania metody z ogromnymi możliwościami, jakie zapewnia zdalne wywoływanie metod.

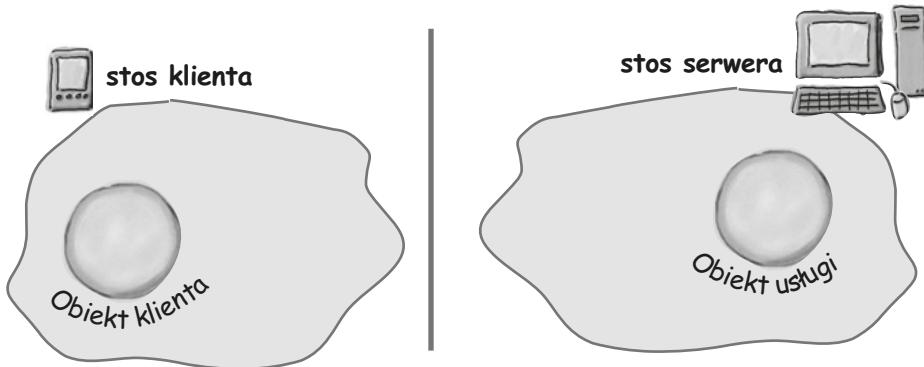
Właśnie te możliwości daje nam *RMI* (ang. *Remote Method Invocation*)!

Cofnijmy się jednak i zastanówmy, w jaki sposób zaprojektowalibyśmy RMI, gdybyśmy musieli zrobić to samodzielnie. Zrozumienie rozwiązań, które musielibyśmy stworzyć, pomoże nam w poznaniu zasad działania RMI.

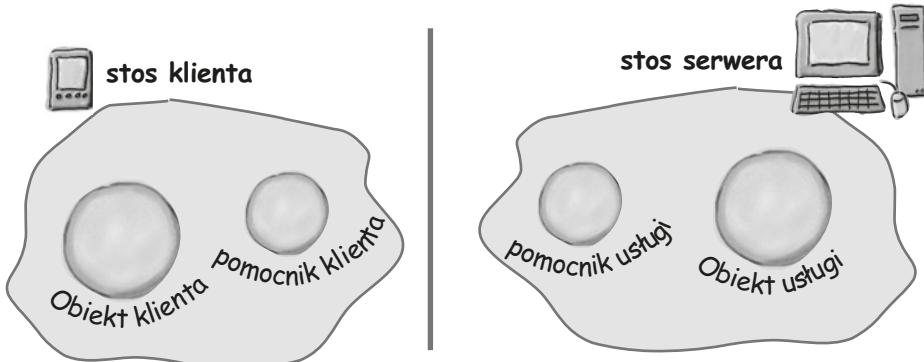
Projekt wywoływania zdalnych metod

Rozwiązanie będzie się składać z czterech elementów: serwera, klienta, pomocnika serwera oraz pomocnika klienta.

- 1 Stwórz aplikacje klienta i serwera. Aplikacja serwera jest **usługą zdalną** posiadającą obiekt udostępniający metodę, którą klient chce wywołać.



- 2 Stwórz obiekty „pomocników” dla aplikacji serwera i klienta. Programy te będą obsługiwać wszystkie szczegóły niskiego poziomu, dzięki czemu klient oraz usługa działająca na serwerze będą mogły udawać, że działają na tej samej stercie.



Przeznaczenie „pomocników”

„Pomocnicy” są obiektami odpowiedzialnymi za komunikację. To właśnie one sprawiają, że klient może działać w taki sposób, jak gdyby wywoływał metody lokalnego obiektu. W rzeczywistości, tak właśnie się dzieje. Klient wywołuje metodę swojego pomocnika, jak gdyby pomocnik był faktyczną usługą. Jednak pomocnik klienta jest jedynie pośrednikiem zapewniającym dostęp do „prawdziwej usługi”.

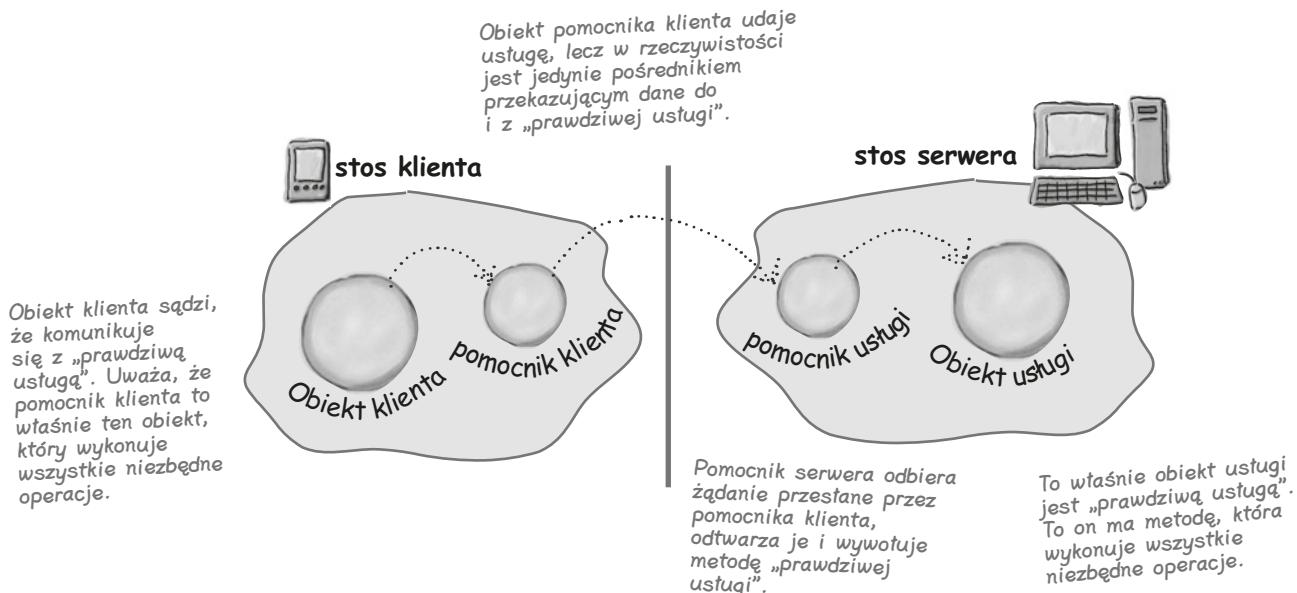
Innymi słowy, obiekt klienta sądzi, że wywołuje metodę obiektu zdalnego, gdyż pomocnik klienta udaje, że jest obiektem usługi. *Czyli udaje, że jest obiektem, którego metodę klient chce wywołać!*

Jednak pomocnik klienta tak naprawdę nie jest zdalną usługą. Choć działa tak jak ona (gdyż udostępnia tę samą metodę, którą ma mieć usługa), to jednak nie implementuje logiki jej działania. Zamiast faktycznej metody pomocnik kontaktuje się z serwerem, przekazuje do niego informacje (czyli nazwę metody oraz argumenty jej wywołania) i oczekuje na wyniki przesłane z serwera.

Z kolei po stronie serwera pomocnik serwera odbiera żądanie przesłane przez pomocnika klienta (za pośrednictwem połączenia sieciowego), odtwarza informacje dotyczące wywołania, po czym wywołuje prawdziwą metodę prawdziwego obiektu usługi. A zatem z punktu widzenia obiektu usługi, wywołanie jest lokalne, gdyż zostało wykonane przez pomocnika, a nie przez zdalny obiekt klienta.

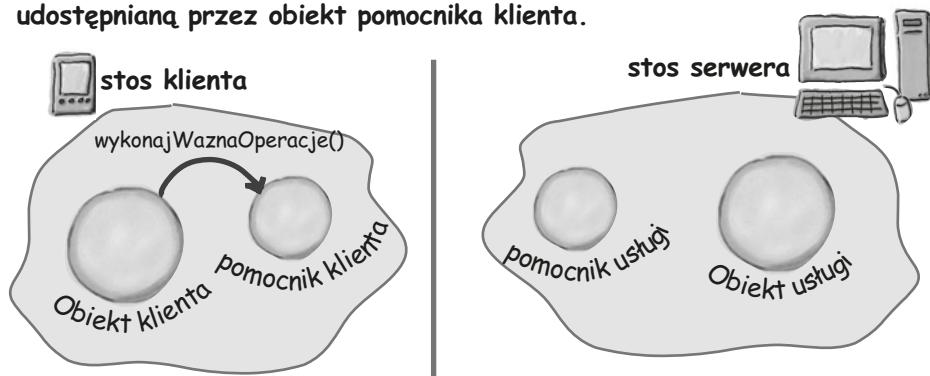
Pomocnik usługi odbiera wartość wynikową zwróconą przez usługę, przygotowuje ją i przesyła z powrotem do pomocnika klienta (wykorzystując w tym celu strumień wyjściowy gniazda). Pomocnik klienta odtwarza przesłane informacje i przekazuje wartość wynikową do obiektu klienta.

Obiekt klienta chce działać w taki sposób, jak gdyby wywoływał zdalne metody. Jednak w rzeczywistości wywołuje lokalny obiekt „pośredniczący”, który obsługuje wszystkie szczegóły związane z wymianą informacji przy użyciu gniazd i strumieni.

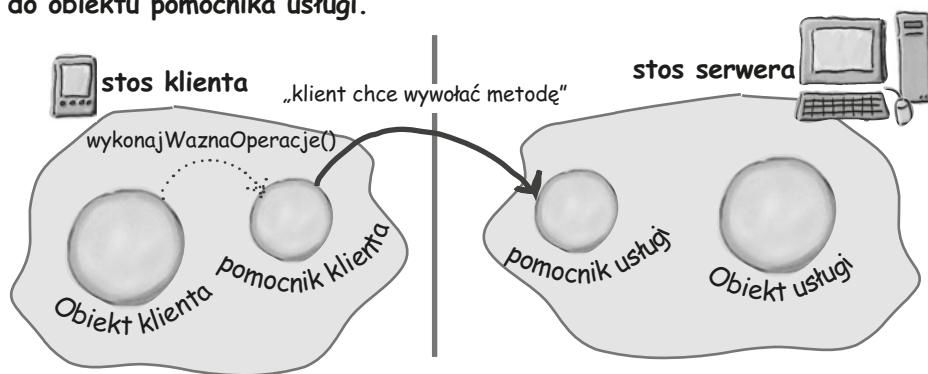


Oto w jaki sposób jest wywoływana metoda:

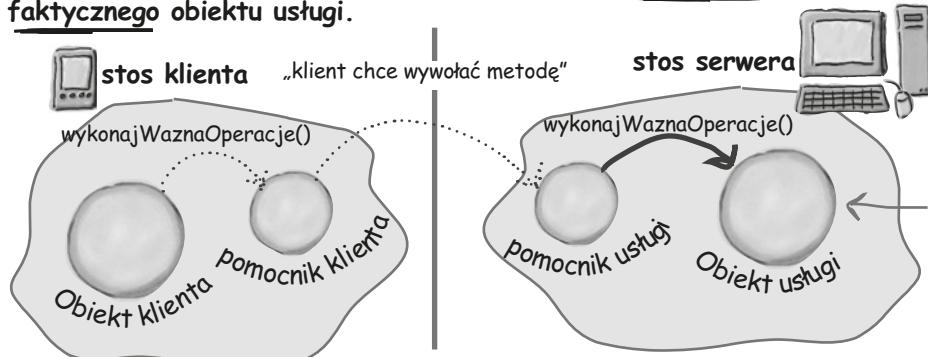
- Obiekt klienta wywołuje metodę `wykonajWaznaOperacje()` udostępnianą przez obiekt pomocnika klienta.



- Obiekt pomocnika klienta pakuje informacje o wywołaniu (argumenty, nazwę metody itd.) i przesyła je siecią do obiektu pomocnika usługi.



- Obiekt pomocnika serwera odpakowuje informacje przesłane przez pomocnika klienta, określa, jaką metodę należy wywołać (i który obiekt udostępnia tę metodę), po czym wywołuje faktyczną metodę faktycznego obiektu usługi.



Pamiętaj, że właściwie to jest obiekt, w którym została zaimplementowana faktyczna logika metody. To właściwie on robi to, o co nam chodzi!

RMI udostępnia obiekty pomocnicze klienta i serwera

W Javie RMI tworzy obiekty pomocników klienta i serwera, a nawet wie, jak sprawić, aby obiekt pomocnika klienta wyglądał tak jak „prawdziwa usługa”. Innymi słowy, RMI wie, w jaki sposób udostępnić w obiekcie pomocnika klienta te same metody, które posiada usługa i które chcemy wywoływać.

Co więcej, RMI udostępnia całą infrastrukturę zapewniającą poprawne działanie aplikacji, w tym także usługę, która pozwala klientowi na odszukanie i utworzenie niezbędnego obiektu pomocnika (pośrednika zapewniającego komunikację z „prawdziwą usługą”).

Dzięki możliwościom, jakie zapewnia RMI, nie musimy sami pisać *nawet wiersza kodu* związanego z obsługą komunikacji sieciowej lub operacji wejścia-wyjścia. Klient może wywoływać zdalne metody (czyli te udostępniane przez „prawdziwą usługę”) zupełnie tak samo, jak gdyby to były lokalne wywołania metod udostępnianych przez obiekty znajdujące się na lokalnej stercie.

No... prawie.

Istnieje jedna różnica pomiędzy wywołaniami RMI oraz wywołaniami lokalnych (zwyczajnych) metod. Otóż należy pamiętać, że chociaż z punktu widzenia klienta wydaje się, iż są to wywołania lokalne, to w rzeczywistości pomocnik klienta przekazuje je siecią. A zatem, ich wykonanie wiąże się z wykonaniem operacji sieciowych oraz operacji wejścia-wyjścia. A co możemy powiedzieć o metodach wykonujących takie operacje?

Są ryzykowne!

W RMI pomocnik klienta określany jest jako „POŚREDNIK” lub „NAMIASTKA”, a pomocnik serwera jako „SZKIELET”.



Zgłaszą całą masę wyjątków.

A zatem klient musi potwierdzić, że zdaje sobie sprawę z tego ryzyka. Musi potwierdzić, że wywołując zdalną metodę, choć wydaje się, że jest to lokalne wywołanie, to w efekcie wiąże się z wykonaniem operacji sieciowych oraz operacji wejścia-wyjścia. Początkowe wywołanie wykonywane przez klienta jest wywołaniem *lokalnym*, ale obiekt pomocnika zamienia je na wywołanie *zdalne*. A zdalne wywołanie oznacza, że wywoływana jest metoda obiektu wykonywanego przez inną wirtualną maszynę Javy. Sposób, w jaki informacje o wywołaniu zostają przekazane z jednej wirtualnej maszyny Javy na drugą, zależy od protokołu używanego przez obiekty pomocników.

Technologia RMI udostępnia dwa protokoły: JRMP oraz IIOP. JRMP to „rodzimy” protokół technologii RMI stworzony w celu zdalnego wywoływanego metod napisanych w Javie przez kod także napisany w Javie. Z kolei protokół IIOP jest stosowany w technologii CORBA (ang. *Common Object Request Broker Architecture*) i pozwala na wywoływanie metod obiektów, które nie muszą być napisane w Javie. Stosowanie technologii CORBA zazwyczaj jest znacznie bardziej skomplikowane niż stosowanie RMI, gdyż aplikacje klienta i serwera nie muszą być napisane w Javie, a to zmusza do wykonywania bardzo wielu przekształceń i konwersji danych.

Na szczęście interesują nas tylko aplikacje pisane w Javie, więc skoncentrujemy się na zwykłe, starej i dosyć prostej technologii RMI.

Tworzenie zdalnej usługi

Poniżej przedstawiony został ogólny opis pięciu etapów tworzenia zdalnej usługi (uruchamianej na serwerze). Nie przejmuj się, wszystkie te czynności zostaną szczegółowo opisane na kolejnych stronach.



Etap pierwszy

Stworzenie zdalnego interfejsu.

Zdalny interfejs określa metody, które klient może zdalnie wywoływać. To właśnie on będzie wykorzystywany przez klienta jako polimorficzny typ reprezentujący obiekt usługi. Interfejs ten zostanie zaimplementowany zarówno w pośredniku, jak i faktycznej usłudze!



Ten interfejs definiuje zdalne metody, które klient będzie mógł wywoływać.

Etap drugi

Stworzenie implementacji zdalnego interfejsu

To właśnie ta klasa wykonuje wszystkie najważniejsze operacje. Implementuje ona metody zdefiniowane w zdalnym interfejsie, a klient chce wywoływać metody obiektu tej klasy.



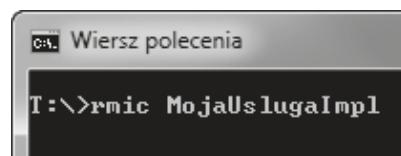
To właśnie jest „prawdziwa usługa”. Klasa zawierająca metody realizujące czynności, na których nam zależy. Ta klasa implementuje zdalny interfejs.

Etap trzeci

Generacja pośrednika i szkieletu przy użyciu programu rmic

To są „pomocnicy” klienta i serwera. Nie musisz sam tworzyć tych klas, ani nawet zaglądać do ich wygenerowanego kodu źródłowego. Są one tworzone całkowicie automatycznie przez program rmic wchodzący w skład JDK.

powoduje wygenerowanie dwóch nowych klas reprezentujących obiekty pomocnicze.



MojaUslugaImpl_Stub.class

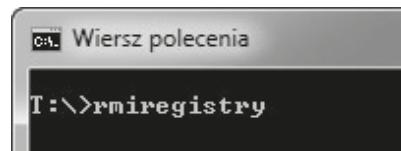


MojaUslugaImpl_Skel.class

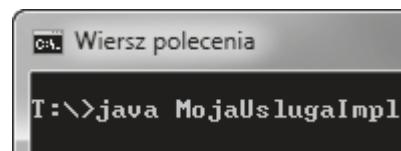
Etap czwarty

Uruchomienie rejestru RMI (rmiregistry)

Rejestr RMI to jak gdyby puste, białe karty książki telefonicznej. Do tego rejestru zgłasza użytkownik, chcąc zdobyć obiekt pośrednika (obiekt pomocnika klienta).



Uruchom ten program w osobnym oknie poleceń (terminalu)



Etap piąty

Uruchomienie zdalnej usługi

Musisz stworzyć i uruchomić obiekt usługi. Trzeba utworzyć obiekt klasy implementującej usługę i zarejestrować go w rejestrze RMI. Zarejestrowanie usługi sprawia, że staje się ona dostępna dla klientów.

Etap pierwszy. Stworzenie zdalnego interfejsu



1 Stwórz interfejs rozszerzający java.rmi.Remote

Remote to interfejs „znacznik”, co oznacza, że nie definiuje on żadnych metod. Jednak z punktu widzenia RMI interfejs ten ma specjalne znaczenie, a zatem konieczne jest przestrzeganie tej reguły. Zwróć uwagę na użycie słowa kluczowego extends. Otóż dozwolone jest, aby jeden interfejs rozszerzał drugi.

```
public interface MojaUsluga extends Remote {
```

Twój interfejs musi informować, że służy do wywoływania zdalnych metod. Interfejs nie może niczego implementować, lecz może rozszerzać inne interfejsy.

2 W deklaracjach metod zaznacz, że mogą one zgłaszać wyjątki RemoteException

Zdalny interfejs jest używany przez klienta jako polimorficzny typ pozwalający na odwoływanie się do metod usługi. Innymi słowy, klient wywołuje metody jakiegoś obiektu, który implementuje ten interfejs. Obiektem tym jest oczywiście pośrednik klienta, a ponieważ realizuje ona wszelkiego typu operacje sieciowe oraz operacje wejścia/wyjścia, zatem podczas ich wykonywania może się wydarzyć wiele nieprzewidzianych sytuacji. Klient musi potwierdzić świadomość ryzyka, deklarując lub obsługując odpowiednie wyjątki. Jeśli metody wchodzące w skład interfejsu deklarują możliwość zgłaszania wyjątków, to każdy kod wywołujący te metody przy użyciu odwołania odpowiedniego typu (typu interfejsu) także musi deklarować te wyjątki lub je obsługiwać.

```
import java.rmi.*;           ← Interfejs Remote należy do pakietu java.rmi.
public interface MojaUsluga extends Remote {
    public String powiedzCzesc() throws RemoteException;
}
```

Każde wywołanie zdalnej metody jest uważane za przedsięwzięcie „ryzykowne”. Zadeklarowanie wyjątku RemoteException w każdej z metod zmusi klienta, by zwrócić uwagę na to niebezpieczeństwo i potwierdzić świadomość faktu, że kod może nie zadziałać.

3 Upewnij się, że argumenty i wartości wynikowe są wartościami typów podstawowych lub klas implementujących interfejs Serializable

Argumenty i wartości wynikowe zdalnych metod muszą być wartościami typów podstawowych lub obiektami klas, które implementują interfejs Serializable. Zastanów się nad tym. Każdy argument przekazywany do zdalnej metody musi zostać spakowany i przesłany siecią, a to „pakowanie” jest możliwe dzięki wykorzystaniu mechanizmu serializacji. Dokładnie to samo dotyczy wartości wynikowych. Jeśli będziesz używać wartości typów podstawowych, łańcuchów znaków lub większości typów dostępnych w API (w tym także tablic i kolekcji), to nie powinieneś mieć żadnych problemów. Jeśli jednak przekazujesz własne obiekty, to upewnij się, że ich klasy implementują interfejs Serializable.

```
public String powiedzCzesc() throws RemoteException;
```

↑
Ta wartość zostanie przesłana siecią z serwera do klienta, a zatem musi zostać serializowana. To właśnie w ten sposób przesyłane są wartości argumentów oraz wyniki metod.

Etap drugi. Stworzenie implementacji zdalnego interfejsu



1 Zaimplementuj interfejs Remote

Twoja usługa musi implementować zdalny interfejs — czyli interfejs zawierający metody, które będzie wywoływać klient.

```
public class MojaUslugaImpl extends UnicastRemoteObject implements MojaUsluga {
    public String powiedzCzesc() {
        return "Serwer mówi: 'Czołem!'";
    }
} // dalszy kod klasy
```

Kompilator upewni się, że zaimplementowałeś wszystkie metody implementowanego interfejsu. W naszym przypadku jest tylko jedna taka metoda.

2 Rozszerz klasę UnicastRemoteObject

Aby Twój obiekt mógł działać jako obiekt zdalnej usługi, będzie potrzebował pewnych możliwości funkcjonalnych związanych z faktem, iż jest „zdalny”. Dla Ciebie najprostszym sposobem zapewnienia mu tych możliwości jest rozszerzenie klasy UnicastRemoteObject (należącej do pakietu `java.rmi.server`).

```
public class MojaUslugaImpl extends UnicastRemoteObject implements MojaUsluga {
```

3 Stwórz konstruktor bezargumentowy deklarujący wyjątek RemoteException

Twoja nowa klasa bazowa, `UnicastRemoteObject`, ma jeden mały problem — jej konstruktor zgłasza wyjątek `RemoteException`. Jedynym sposobem rozwiązania tego problemu jest stworzenie konstruktora — wyłącznie po to, aby zadeklarować w nim wyjątek `RemoteException`. Pamiętaj, że podczas tworzenia obiektu zawsze wywoływany jest konstruktor klasy bazowej. Jeśli konstruktor klasy bazowej zgłasza wyjątek, to nie masz innego wyjścia jak zadeklarować, że także Twój konstruktor może zgłaszać ten wyjątek.

```
public MojaUslugaImpl() throws RemoteException {}
```

W konstruktorze nie musisz umieszczać żadnego kodu. Konstruktor stanowi jedynie sposób, aby zadeklarować, że konstruktor klasy bazowej zgłasza wyjątek.

4 Zarejestruj usługę w rejestrze RMI.

Teraz, kiedy już stworzyłeś zdalną usługę, musisz ją udostępnić klientom. Możesz to zrobić poprzez utworzenie obiektu usługi oraz umieszczenie go w rejestrze RMI (który w tym momencie musi już działać, gdyż w przeciwnym przypadku cała operacje się nie powiedzie). Kiedy rejestrujesz obiekt implementujący usługę, system RMI umieszcza w rejestrze *pośrednika*, ponieważ to właśnie jego będzie potrzebował klient. Usługę należy rejestrować poprzez wywołanie statycznej metody `rebind()` klasy `java.rmi.Naming`.

```
try {
    MojaUsluga usluga = new MojaUslugaImpl();
    Naming.rebind("Zdalne czesc", usluga);
} catch (Exception ex) {...}
```

Określ nazwę swojej usługi (która klienci będą mogli używać w celu odszukania usługi w rejestrze) oraz zarejestruj ją w rejestrze RMI. Kiedy skojarzysz obiekt usługi, RMI zamieni usługę na obiekt pośrednika i umieści pośrednika w rejestrze.

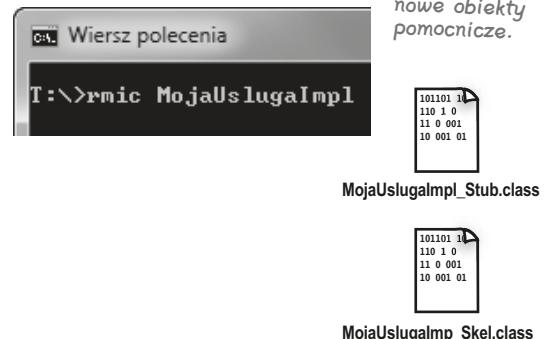
Etap trzeci. Generacja namiastek i szkieletów

- Uruchom program rmic, podając przy tym nazwę klasy implementującej zdalny interfejs (a nie samego interfejsu)

Program narzędziowy rmic dostarczany wraz z JDK pobiera klasę implementującą usługę i na jej podstawie tworzy dwie nowe klasy - pośrednika oraz szkielet. Podczas określania nazw nowych klas program ten wykorzystuje konwencję nazewnictwa, która polega na dodaniu do nazwy wskazanej klasy końcówek: _Stub oraz _Skel. Program rmic daje także inne możliwości, takie jak pominięcie generacji szkieletu, podgląd kodu źródłowego generowanych klas, a nawet, wykorzystanie protokołu IIOP. Jednak opisane czynności prezentują najczęściej stosowany sposób tworzenia namiastek i pośredników. Klasy wygenerowane przez program rmic zostaną umieszczone w bieżącym katalogu (czyli w dowolnym katalogu, do którego przeszedłeś). Pamiętaj, że program musi mieć dostęp do klasy implementującej usługę, zatem najprawdopodobniej będziesz go wywoływał z katalogu, w którym ta klasa się znajduje. (W tym przykładzie celowo nie używamy pakietów, aby jak najbardziej uprościć proces tworzenia aplikacji RMI, jednak w praktyce zapewne będziesz musiał uwzględnić strukturę katalogów pakietu oraz stosować w pełni kwalifikowane nazwy klas).

Zauważ, że w poleceniu nie jest podawane rozszerzenie „.class”, a jedynie nazwa klasy.

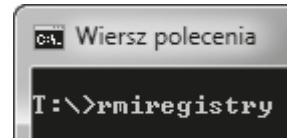
W wyniku generowane są dwa nowe obiekty pomocnicze.



Etap czwarty. Uruchomienie programu rmiregistry

- Wyświetl okno wiersza poleceń i uruchom program rmiregistry

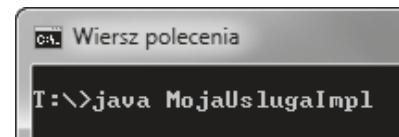
Pamiętaj, aby uruchomić ten program z katalogu, który ma dostęp do Twoich klas. Najprostszym rozwiązaniem będzie uruchomienie go z katalogu plikiklasowe.



Etap piąty. Uruchomienie usługi

- Uruchom kolejne okno wiersza poleceń i uruchom usługę

Mozna to zrobić, na przykład, uruchamiając metodę main() klasy implementującej usługę. W naszym bardzo prostym przykładzie, cały kod inicjujący usługę został umieszczony w metodzie, która ją implementuje; metoda main() tej klasy tworzy obiekt usługi i rejestruje go w rejestrze RMI.



Pełny kod aplikacji działającej po stronie serwera



Zdalny interfejs:

```
import java.rmi.*;           Klasa RemoteException oraz interfejs
                             Remote należą do pakietu java.rmi.

public interface MojaUsluga extends Remote {           Twój interfejs MUSI rozszerzać
                                                       interfejs java.rmi.Remote.

    public String powiedzCzesc() throws RemoteException;
}

}                                                       Wszystkie zdalne metody muszą
                                                       deklarować wyjątek RemoteException.
```

Zdalna usługa (implementacja):

```
import java.rmi.*;           UnicastRemoteObject należy
import java.rmi.server.*;     do pakietu java.rmi.server.

public class MojaUslugaImpl extends UnicastRemoteObject implements MojaUsluga {           Rozszerzenie klasy UnicastRemoteObject
                                                       jest najprostszym sposobem stworzenia
                                                       zdalnego obiektu.

    public String powiedzCzesc() {           Oczywiście trzeba
        return "Serwer mówi: 'Czołem!'";     zaimplementować wszystkie
                                                       metody interfejsu. Zwróć jednak
                                                       uwagę, że nie musisz deklarować
                                                       wyjątku RemoteException.

    }

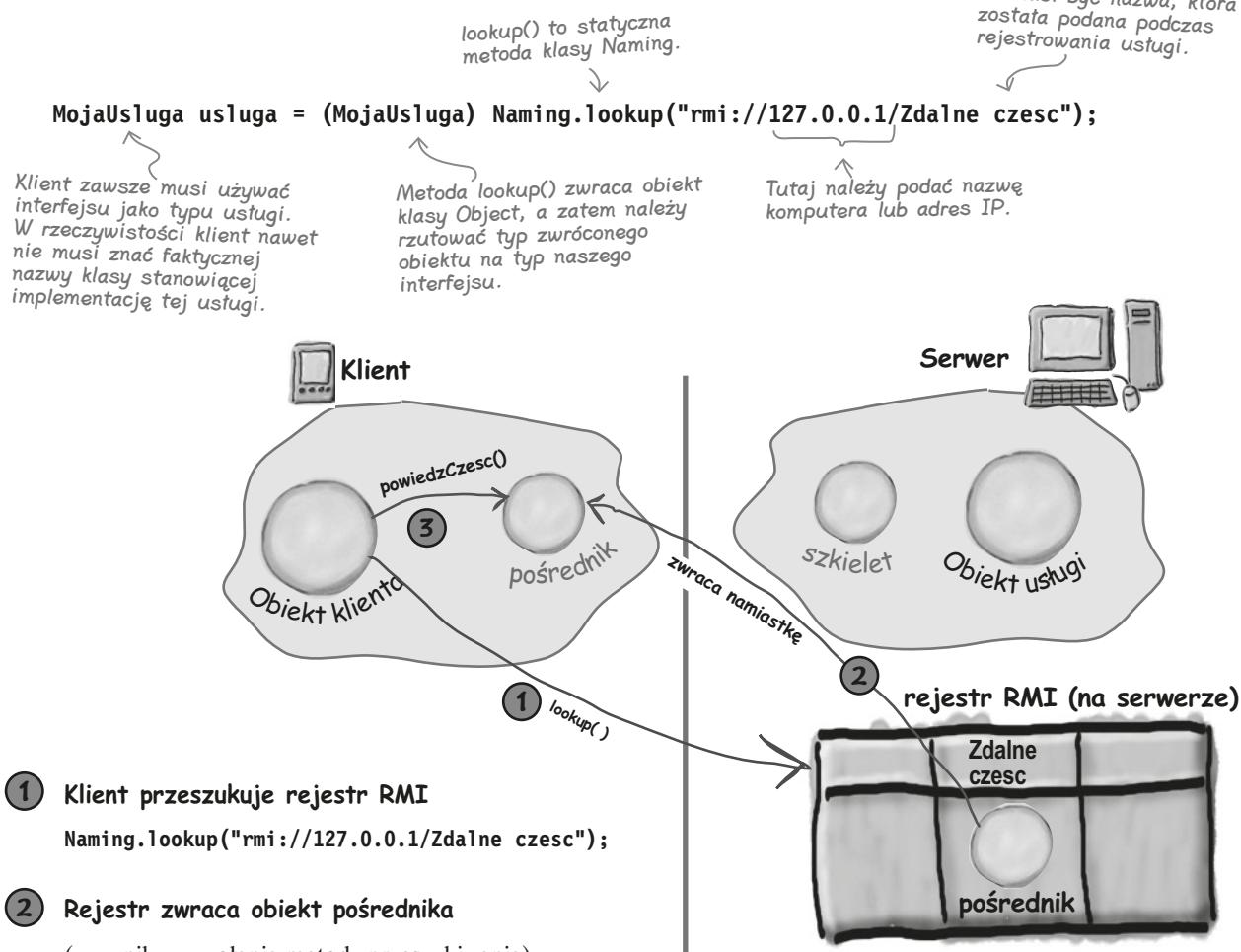
    public MojaUslugaImpl() throws RemoteException { }           MUSISZ zaimplementować
                                                               zdalny interfejs!!!

    public static void main(String[] args) {           Konstruktor klasy bazowej (UnicastRemoteObject)
        try {                                         deklaruje wyjątek; oznacza to, że tworząc obiekt,
            MojaUsluga usluga = new MojaUslugaImpl();   wykonyujesz ryzykowny kod i z tego powodu
            Naming.rebind("Zdalne czesc", usluga);       w swojej klasie musisz stworzyć konstruktor.

        } catch (Exception ex) {           Utwórz zdalny obiekt, a następnie „skojarz” go
            ex.printStackTrace();                   z rejestrzem RMI przy wykorzystaniu metody
                                                       Naming.rebind(). Nazwa podana podczas
                                                       rejestracji będzie wykorzystywana przez
                                                       klientów podczas przeszukiwania rejestrów.
        }
    }
}
```

W jaki sposób klient zdobywa obiekt Pośrednika?

Klient musi zdobyć obiekt pośrednika, gdyż to właśnie ten obiekt będzie używany do wywoływania metody. I właśnie w tym miejscu na arenę wkracza rejestr RMI. Klient „przeszukuje” rejestr, podobnie jak przeglądałby karty książki telefonicznej, i w końcu stwierdza: „A oto mamy nazwę, teraz chciałbym dostać obiekt pośrednika dołączony do tej nazwy”.



- 1 Klient przeszukuje rejestr RMI

`Naming.lookup("rmi://127.0.0.1/Zdalne czesc");`

- 2 Rejestr zwraca obiekt pośrednika

(w wyniku wywołania metody przeszukiwania) i automatycznie przeprowadza jego deserializację. Klasa pośrednika (ta wygenerowana przez program rmic) MUSI być dostępna na komputerze klienta, gdyż w przeciwnym razie nie uda się deserializować obiektu.

- 3 Klient wywołuje metodę, używając w tym celu obiektu pośrednika, jak gdyby właśnie ten obiekt był usługą

W jaki sposób klient zdobywa klasę pośrednika?

Teraz dochodzimy do interesującego zagadnienia. W jakiś tajemniczy sposób klient musi zdobyć klasę pośrednika (wygenerowaną wcześniej przy użyciu programu rmic). Klasa ta musi być dostępna w czasie, gdy klient przeszukuje rejestr RMI, gdyż w przeciwnym razie nie będzie można deserializować pośrednika i cała nasza rozproszona aplikacja nie będzie działać. W przypadku prostych systemów można ręcznie dostarczyć klasę pośrednika użytkownikowi końcowemu.

Istnieje jednak znacznie ciekawsze rozwiązanie, choć wykracza ono poza ramy niniejszej książki. Na wszelki wypadek, gdybyś był zainteresowany tym zagadnieniem podajemy, że rozwiązanie to nosi nazwę „dynamicznego ładowania klas”. W przypadku zastosowania dynamicznego ładowania klas pośrednik (lub jakikolwiek inny obiekt, który można serializować) zostaje „opatrzony” adresem URL, który informuje system RMI na komputerze klienta, gdzie może znaleźć plik klasowy danego obiektu. Następnie, podczas procesu deserializacji obiektu, jeśli RMI nie jest w stanie znaleźć odpowiedniego pliku klasowego na lokalnym komputerze, pobiera go przy użyciu protokołu HTTP (i żądania GET) z miejsca określonego podanym adresem URL. A zatem, abyś mógł udostępniać pliki klasowe, wystarczy zainstalować zwyczajny serwer WWW; prawdopodobnie będziesz także musiał zmienić niektóre parametry zabezpieczeń na komputerze, na którym działa klient. Istnieje jeszcze kilka ciekawych sztuczek związanych z dynamicznym ładowaniem klas, jednak my możemy przedstawić jedynie ogólny zarys tego rozwiązania.

Pełny kod klienta

```
import java.rmi.*; ← Klasa Naming (umożliwiająca
                      przeszukiwanie rejestrów RMI)
                      należy do pakietu java.rmi.
```

```
public class MojaUslugaKlient {
    public static void main(String[] args) {
        new MojaUslugaKlient().doRoboty();
    }

    public void doRoboty() {
        try {
            MojaUsluga usluga = (MojaUsluga) Naming.lookup("rmi://127.0.0.1/Zdalne czesc");
            String s = usluga.powiedzCzesc();
            System.out.println(s);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Obiekt jest zwracany z rejestrów RMI w formie obiektu klasy Object, a zatem nie zapomnij odpowiednio rzutować jego typu.

Potrzebujesz adresu IP lub nazwy komputera

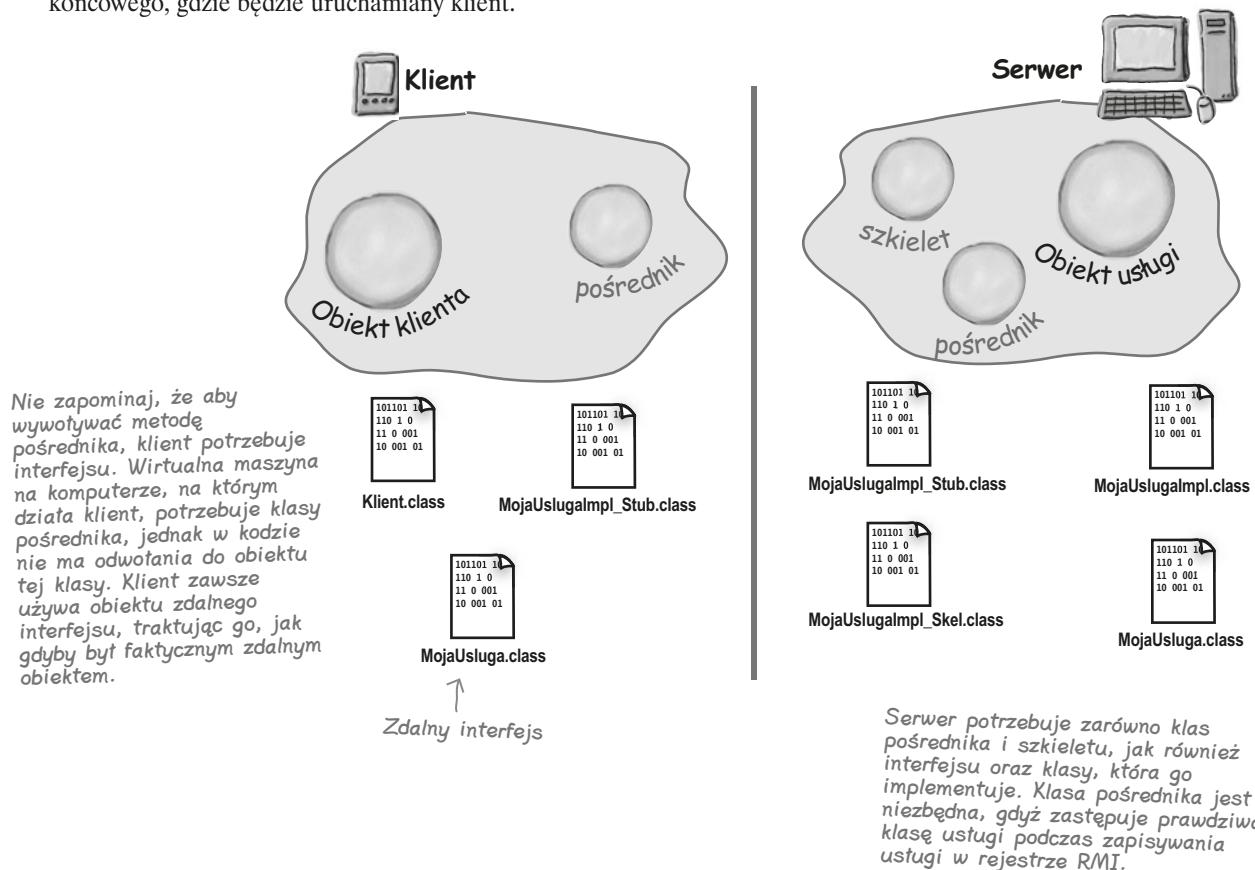
To wygląda jak stare, zwykłe wywołanie metody! (Oprócz tego, że wymaga obsługi wyjątku RemoteException).

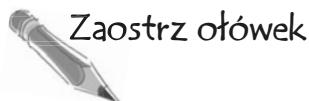
↑
oraz nazwy podanej podczas rejestracji usługi.

Upewnij się, że na każdym komputerze będą niezbędne pliki klasowe

Oto trzy podstawowe błędy, jakie popełniają programiści stosujący technologię RMI:

1. Zapominają uruchomić program rmiregistry przed uruchomieniem zdalnej usługi (w chwili rejestracji usługi przy użyciu metody Naming.rebind(), program rmiregistry musi już działać!).
2. Zapominają, że typy argumentów oraz wartości wynikowych muszą pozwalać na serializację (ale o tym przekonamy się dopiero podczas działania programu, gdyż kompilator nie jest w stanie tego wykryć).
3. Zapominają umieścić plik klasowy pośrednika na komputerze użytkownika końcowego, gdzie będzie uruchamiany klient.





Co było pierwsze



Spójrz na przedstawioną poniżej sekwencję zdarzeń i rozmieśc je według kolejności, w jakiej zachodzą w aplikacji RMI.



CELNE SPOSTRZEŻENIA

- Obiekt na stercie nie może zdobyć normalnego odwołania do obiektu znajdującego się na innej stercie (czyli wykonywanego przez inną wirtualną maszynę Javy).
 - Technologia RMI (ang. *Remote Method Invocation*) sprawia, że mamy wrażenie wywoływania metody zdalnego obiektu (czyli obiektu wykonywanego przez inną wirtualną maszynę Javy), choć w rzeczywistości tak się nie dzieje.
 - Kiedy klient wywołuje metodę zdalnego obiektu w rzeczywistości odwołuje się do lokalnego obiektu pośredniczącego, który przekazuje wywołanie do obiektu zdalnego. Ten obiekt pośredniczący nazywany jest „pośrednikiem” lub „namiastką”.
 - Pośrednik jest obiektem pomocniczym klienta, który przygotowuje i przesyła wywołanie metody na serwer, obsługując przy tym wszystkie operacje sieciowe niskiego poziomu (gniazda, strumienie, serializację itp.).
 - Aby stworzyć zdальную usługę (czyli obiekt, którego metoda zostanie w końcu wywołana w celu obsłużenia żądania klienta), musisz zacząć od stworzenia zdalnego interfejsu.
 - Zdalny interfejs musi rozszerzać interfejs `java.rmi.Remote`, a wszystkie jego metody muszą deklarować wyjątek `RemoteException`.
 - Zdalna usługa implementuje zdalny interfejs.
 - Zdalna usługa powinna rozszerzać klasę `UnicastRemoteObject`. (Z technicznego
- punktu widzenia są także inne metody stworzenia zdalonego obiektu, jednak wykorzystanie klasy `UnicastRemoteObject` jest najprostszą z nich).
- Twoja klasa zdalnej usługi musi mieć konstruktor, a konstruktor musi deklarować wyjątek `RemoteException` (ponieważ wyjątek ten jest zadeklarowany w konstruktorze klasy bazowej).
 - Obiekt zdalnej usługi musi zostać utworzony i zarejestrowany w rejestrze RMI.
 - Usługę można zarejestrować, wywołując statyczną metodę `Naming.rebind("nazwa usługi", obiektUsługi);`
 - Rejestr RMI musi działać na tym samym komputerze, na którym działa zdalna usługa, przy czym należy go uruchomić przed próbą zarejestrowania obiektu usługi.
 - Klient przeszukuje rejestr RMI, używając w tym celu statycznej metody `Naming.lookup("rmi://nazwaHosta/nazwa usługi");`
 - Niemal wszystkie operacje związane z wykorzystaniem RMI mogą zgłaszać wyjątki `RemoteException` (sprawdzane przez kompilator). Dotyczy to także rejestrowania usług, przeszukiwania rejestrów oraz wszystkich wywołań zdalnych metod wykonywanych za pomocą pośrednika.

Ech... ale kto tak naprawdę używa RMI?

My używamy RMI w naszym nowym, świetnym systemie wspomagania decyzji.



Styszałam, że twoja była żona wciąż używa zwykłych połączeń sieciowych.

Ja używam RMI w poważnej aplikacji biznesowej do prowadzenia handlu elektronicznego wykorzystującej technologię JEE.

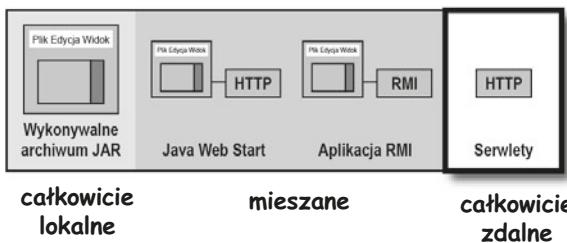


A my mamy system rezerwacji hotelowej bazujący na EJB, a technologia EJB korzysta z RMI.

Po prostu nie jestem w stanie wyobrazić sobie życia bez sieci domowej i urządzeń wykorzystujących technologię Jini.

Ja również! Jak ktokolwiek może się bez niej obejść! Wprost uwielbiam RMI za to, że dzięki niej powstała technologia Jini.





A co z serwletami?

Serwlety to napisane w Javie programy działające na serwerze WWW (i współpracujące z nim). Kiedy użytkownik, wykorzystując przeglądarkę WWW, prowadzi interakcję ze stroną WWW, na serwer przesyłane jest żądanie. Jeśli obsługa tego żądania wymaga pomocy serwletu, to serwer WWW uruchamia kod tego serwletu (lub wywołuje go, jeśli serwlet już działa). Kod serwletu, to zwyczajny kod napisany w Javie, który jest wykonywany przez serwer WWW w celu wykonania czynności koniecznych do obsługi żądania (czynności te mogą polegać, na przykład, na zapisaniu informacji w pliku tekstowym lub bazie danych na serwerze). Jeśli znasz się na skryptach CGI pisanych w Perl-u, to doskonale wiesz, co mamy na myśli. Programiści aplikacji internetowych używają skryptów CGI i serwletów do wykonywania wszelkich możliwych czynności — zaczynając od zapisywania informacji przesyłanych przez użytkowników, a kończąc na obsłudze internetowych list dyskusyjnych.

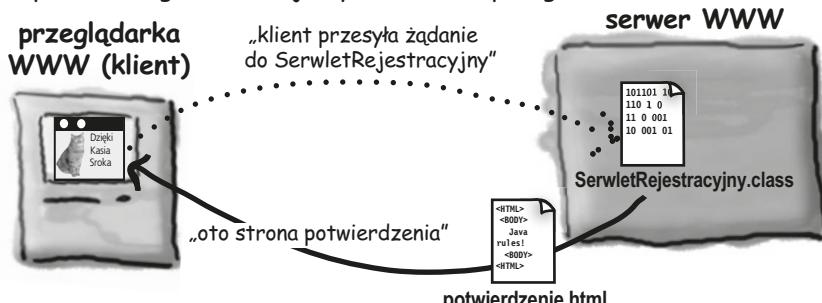
Ale nawet serwlety mogą używać technologii RMI!

Zdecydowanie najczęściej wykorzystywanym zastosowaniem technologii JEE jest łączenie serwletów z komponentami EJB (ang. Enterprise JavaBeans), przy czym serwleты pełnią rolę klientów korzystających z możliwości komponentów EJB. Właśnie w takich sytuacjach *serwlety wykorzystują RMI do komunikowania się z komponentami EJB*. (Choć sposoby wykorzystywania RMI w komponentach EJB są *nieco* inne niż w przykładzie przedstawionym w tym rozdziale).

- 1 Użytkownik wypełnia formularz rejestracyjny i kliką przycisk Wyślij. Serwer WWW odbiera żądanie, określa, że dotyczy ono serwletu, i przesyła żądanie dalej do odpowiedniego serwletu.



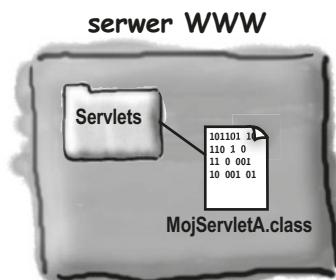
- 2 Serwlet (kod napisany w Javie) zostaje uruchomiony, dodaje dane do bazy danych, generuje stronę WWW (zawierającą informacje dostosowane do żądania) i przesyła je z powrotem na komputer użytkownika, gdzie zostają wyświetlane w przeglądarce.



Etapy tworzenia i uruchamiania serwletu

1 Określ, gdzie należy umieścić serwlet

W tym przykładzie założymy, że dysponujesz już zainstalowanym i działającym serwerem WWW, oraz że serwer został skonfigurowany w sposób umożliwiający korzystanie z serwletów. Najważniejszym zadaniem jest określenie, w którym miejscu mają być umieszczane pliki klasowe serwletów, aby serwer mógł je odnaleźć. Jeśli Twoją witrynę obsługuje jakiś dostawca usług internetowych, to zapewne będzie w stanie udzielić informacji, gdzie należy umieszczać serwlety, podobnie jak może wskazać miejsce przeznaczone do umieszczania skryptów CGI.



2 Odszukaj archiwum `servlets.jar`, a ścieżkę dostępu do niego dodaj do zmiennej środowiskowej `CLASSPATH`

Serwlety nie wchodzą w skład standardowych bibliotek Javy, a zatem będziesz potrzebował pliku `servlets.jar`, w którym są umieszczone wszystkie klasy związane z obsługą serwletów. Możesz go skopiować z witryny `oracle.com` lub znaleźć wśród plików jakiegoś serwera WWW dającego możliwości korzystania z Javy (takiego jak Tomcat, który można znaleźć pod adresem `jakarta.apache.org`). Bez dostępu do tych klas nie będziesz w stanie skompilować swoich serwletów.



3 Stwórz klasę serwletu, rozszerzając klasę `HttpServlet`

Serwlet to zwyczajna klasa Javy, która rozszerza klasę `HttpServlet` (należącą do pakietu `javax.servlet.http`). Można tworzyć także inne typy serwletów, jednak w większości przypadków będą nas interesować wyłącznie serwlety typu `HttpServlet`.

```
public class MojServletA extends HttpServlet { ... }
```



4 Napisz stronę WWW odwołującą się do serwletu

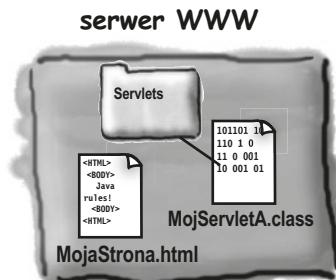
Kiedy użytkownik klinie łącze odwołujące się do serwletu, serwer WWW odszuka serwlet i w zależności od polecenia HTTP (GET, POST itd.) wywoła jego odpowiednią metodę.



```
<a href="servlets/MojServletA">Zadziwiający serwlet</a>
```

5 Udostępnij stronę WWW oraz serwlet na serwerze WWW

Ten etap zależy wyłącznie od używanego serwera WWW (a właściwie od używanej wersji technologii Java Servlets). Dostawca usług internetowych może Cię poinstruować, byś po prostu umieścił swoje serwlety w katalogu `servlets` swojej witryny. Jeśli jednak używasz na przykład ostatniej wersji serwera Tomcat, to będziesz musiał wykonać znacznie więcej pracy, by umieścić serwlety (i strony WWW) w odpowiednich miejscach. (Tak się składa, że wydaliśmy książkę także na ten temat).



Bardzo prosty serwlet

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MojServletA extends HttpServlet {
```

Oprócz pakietu wejścia-wyjścia musimy także zainportować dwa pakiety związane z serwletami. Pamiętaj, że pakiety te NIE należą do standardowej biblioteki Javy — trzeba je skopiować niezależnie.

W większości „normalnych” serwletów będzie rozszerzać klasę HttpServlet i przestaniać jedną z jej metod.

Przestoń metodę doGet(), aby obsługiwać proste żądania HTTP typu GET.

Serwer WWW wywołuje tę metodę, przekazując do niej żądanie przesłane przez przeglądarkę (można z niego pobrać różnego typu informacje) oraz obiekt „odpowiedzi”, którego będziesz używać w celu przestania odpowiedzi (strony WWW) do przeglądarki.

```
public void doGet(HttpServletRequest zadanie, HttpServletResponse odpowiedz) throws
ServletException, IOException {
```

W ten sposób informujemy serwer (oraz przeglądarkę), jakiego typu dane będą przesłane z serwera jako wynik wykonania serwletu.

```
odpowiedz.setContentType("text/html");
```

Obiekt odpowiedzi udostępnia nam strumień wyjściowy używany do „zapisywania” informacji, które zostaną przekazane z powrotem do serwera.

```
PrintWriter out = odpowiedz.getWriter();
```

```
String komunikat = "Jeśli to czytasz, to znaczy, że serwlet działa!";
```

```
out.println("<HTML><BODY>");
out.println("<H1>" + komunikat + "</H1>");
out.println("</BODY></HTML>");
out.close();
}
```

To, co „zapisujemy”, to kod strony HTML! Zostanie ona przekazana do serwera, a następnie dalej — do przeglądarki użytkownika — tak jak każda inna strona WWW, choć aż do tej chwili ta konkretna strona na serwerze nie istniała. Innymi słowy, nie ma żadnego odpowiadającego jej pliku .html.

Oto jak wygląda strona WWW

Strona HTML zawierająca połączenie z serwlem

```
<HTML>
<BODY>
<a href="servlet/MojServletA">Zadziwiający serwlet</a>
</BODY>
</HTML>
```

Kliknij tącle, aby wykonać → serwlet.

Zadziwiający serwlet



CELNE SPOSTRZEŻENIA

- Serwlety to klasy Javy, które są w całości wykonywane przez serwer WWW.
- Serwlety przydają się jako sposób wykonywania kodu na serwerze, w odpowiedzi na operacje wykonywane przez użytkownika na stronie WWW. Na przykład, jeśli użytkownik poda jakieś informacje w polach formularza, to serwlet może zapisać te informacje w bazie danych i zwrócić użytkownikowi odpowiednio przygotowaną stronę potwierdzenia.
- Aby skompilować serwlet, będziesz potrzebował odpowiednich pakietów, które są umieszczone w pliku *servlets.jar*. Klasy związane z obsługą serwletów nie wchodzą w skład standardowej biblioteki Javy, dlatego też należy je skopiować z witryny oracle.com lub odszukać wśród plików dostarczanych wraz z serwerem WWW dającym możliwość korzystania z Javy. (Uwaga: Biblioteka ta jest dodana do Java Enterprise Edition — JEE).
- Aby uruchamiać serwlety, musisz dysponować serwerem WWW, który potrafi je wykonywać (przykładem może być serwer Tomcat, który można pobrać z witryny jakarta.apache.org).
- Serwlet należy umieścić w odpowiednim miejscu, jednak miejsce to zależy wyłącznie od używanego serwera WWW. Dlatego przed próbą uruchomienia serwletu musisz się dowiedzieć, gdzie go trzeba umieścić. Jeśli Twój serwer WWW jest obsługiwany przez dostawcę usług internetowych, który daje możliwość stosowania serwletów, to dostawca powinien udzielić Ci informacji o tym, gdzie należy umieścić serwlety.
- Typowy serwlet rozszerza klasę *HttpServlet* i przesyła jedną lub kilka metod tej klasy, na przykład, *doGet()* lub *doPost()*.
- Serwer WWW uruchamia serwlet i, w zależności od odebranego żądania, wywołuje jego odpowiednią metodę (*doGet()* lub inną).
- Serwlet może wygenerować odpowiedź, pobierając z obiektu „odpowiedzi” (przekazanego w wywołaniu metody *doGet()*) strumień wyjściowy *PrintWriter*.
- Serwlet „zapisuje” w odpowiedzi kompletny dokument HTML (wraz ze wszystkimi znacznikami).

Nie istnieja
główne pytania

P: Co to jest JSP i jaki związek z serwletami ma ta technologia?

O: JSP to skrót pochodzący od słów Java Server Pages. Serwer WWW zamienia dokument JSP na serwlet, jednak różnica pomiędzy obydwiema technologiami polega na tym, co musisz stworzyć Ty — programista. W przypadku serwletów tworzysz klasę zawierającą instrukcje generujące kod HTML (oczywiście jeśli tym, co przesyłasz do przeglądarki, mają być dokumenty HTML). W przypadku stosowania JSP robisz dokładnie na odwrót — tworzysz dokument HTML zawierający kod napisany w Javie.

W ten sposób uzyskujesz możliwość tworzenia dynamicznych stron WWW zawierających zwyczajny kod HTML oraz osadzania wewnętrz nich kodu Javy (a także specjalnych znaczników pozwalających na jego wykonywanie), który jest przetwarzany przez serwer WWW podczas obsługi żądania. Innymi słowy, fragment strony jest generowany przez kod Javy podczas obsługi żądania.

Podstawowa zaleta JSP w stosunku do zwyczajnych serwletów polega na tym, iż znacznie łatwiej jest stworzyć stronę WWW w postaci normalnego kodu HTML niż wygenerować ją przy użyciu serii wywołań metody *println()*. Wyobraź sobie stosunkowo złożoną stronę WWW oraz generowanie jej kodu przy użyciu wywołań metody *println()*. Horror!

Jednak w wielu zastosowaniach tworzenie stron JSP nie jest konieczne, gdyż serwlety nie muszą generować dynamicznych odpowiedzi lub odpowiedzi te są na tyle proste, że ich wygenerowanie nie jest wielkim problemem. A poza tym są dostępne serwery WWW, które obsługują serwlety, lecz nie obsługują dokumentów JSP. W takim przypadku jesteś w kropce.

Kolejną zaletą JSP jest możliwość rozdzielenia pracy, gdyż programiści Javy mogą pisać serwlety, a projektanci stron — dokumenty JSP. Jednak jest to tylko pozorna możliwość, ponieważ osoby, które mają tworzyć dokumenty JSP, i tak muszą poznać Javę (oraz dodatkowe znaczniki wykorzystywane przez technologię JSP), a zatem przypuszczenie, że normalni projektanci stron WWW mogliby tworzyć dokumenty JSP, jest nierealistyczne. Przynajmniej trudno je sobie wyobrazić bez wykorzystania specjalnych narzędzi. Ale jest dobra wiadomość — zaczynają się pojawiać narzędzia pozwalające twórcom stron WWW na tworzenie dokumentów JSP bez konieczności pisania niezbędnego kodu Javy.

P: Czy to wszystko, co macie zamiar powiedzieć na temat serwletów? Po tak obszernym przedstawieniu technologii RMI?

O: Tak. RMI jest integralną częścią języka, a wszystkie klasy związane z tą technologią wchodzą w skład standardowej biblioteki Javy. Ani serwlety, ani technologia JSP nie stanowią części języka, są one uważane za standardowe rozszerzenie. RMI można wykorzystywać na każdej z nowoczesnych wirtualnych maszyn Javy, z kolei, do korzystania z serwletów niezbędny jest odpowiednio skonfigurowany serwer WWW oraz specjalny „kontener” serwletów. W ten delikatny sposób chcemy zasugerować, że „zagadnienia te wykraczają poza ramy tematyczne niniejszej książki”. Więcej informacji na temat serwletów i JSP można znaleźć w książce *Java Servlets i Java Server Pages* wydanej przez wydawnictwo Helion.

Tak dla zabawy przeróbmy nasz program krasomówczy na serwlet

Teraz, kiedy już zapowiedzieliśmy, że nie napiszemy niczego więcej o serwletach, nie możemy się oprzeć pokusie serwletyzacji (no tak, *możemy* użyć takiego czasownika) naszego programu krasomówczego przedstawionego w rozdziale 1. Serwlet to przecież zwyczajny kod napisany w Javie, a przecież nic nie stoi na przeszkodzie, aby w pewnym miejscu kodu wywoływać kod umieszczony w innych klasach.

A zatem, nie ma przeszkód, by serwlet wywoływał kod programu krasomówczego. Aby to umożliwić, należy jedynie skopiować plik klasowy naszego programu do katalogu, w którym znajduje się serwlet.

(Kod programu krasomówczego został przedstawiony na następnej stronie).

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletKrasomowczy extends HttpServlet {
    public void doGet (HttpServletRequest zadanie, HttpServletResponse odpowiedz)
                      throws ServletException, IOException {
        PrintWriter out;
        String tytul = "Krasomówca wygenerował następujące zdanie.";
        odpowiedz.setContentType("text/html");
        out = odpowiedz.getWriter();

        out.println("<HTML><HEAD><TITLE>");
        out.println("Krasomówca");
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>" + tytul + "</H1>"); ↓
        out.println("<P>" + Krasomowca.tworzZdanie());
        out.println("<P><a href=\"ServletKrasomowczy\">wygeneruj inne zdanie</a></p>");
        out.println("</BODY></HTML>");
        out.close();
    }
}
```

Widzisz? Twój serwlet może wywoływać metody innych klas. W tym przypadku wywołujemy statyczną metodę tworzZdanie() klasy Krasomowca (jej kod został przedstawiony na następnej stronie).



Wypróbuju mój nowy internetowy program „krasomówczy”, a będziesz mówić równie pięknie i mądrze jak szef lub ci goście z działu marketingu.

Kod „krasomówczy” w wersji przystosowanej do wykorzystania w serwecie

Zamieszczony poniżej kod został nieco zmieniony w porównaniu z programem przedstawionym w rozdziale 1. W oryginalnym programie, cały kod jest umieszczony wewnątrz metody `main()`, a chcąc wygenerować nowe zdanie za każdym razem musimy go uruchamiać z poziomu wiersza poleceń. W wersji przedstawionej poniżej, kasa zawiera statyczną metodę `tworzZdanie()`, której każde wywołania powoduje zwrócenie łańcucha znaków (zawierającego zdanie). Dzięki temu możesz wywołać tę metodę z dowolnego innego kodu i uzyskać łańcuch znaków zawierający losowo wygenerowanie zdanie.

Pamiętaj, że te długie, zajmujące kilka wierszy instrukcje tworzące tablice łańcuchów znaków, zostały w ten sposób sformatowane przez edytor tekstu. Nie dziel sam jednego długiego wiersza, po prostu wpisz go, a używany edytor zrobi resztę. I bez względu na to, co robisz, nigdy nie naciskaj klawisza *Enter* wewnątrz łańcucha znaków (czyli gdzieś pomiędzy dwoma znakami cudzysłowu).

```
public class Krasomowca {  
    public static String tworzZdanie() {  
        // trzy grupy słów, które będą wybierane do zdania (dodaj własne!)  
        String[] listaSlow1 = {"architektura wielowarstwowa", "30000 metrów", "rozwiązanie B-do-B",  
        "aplikacja kliencka", "interfejs internetowy", "karta inteligentna", "rozwiązanie dynamiczne", "sześć  
        sigma", "przenikliwość"};  
  
        String[] listaSlow2 = {"zwiększa możliwości", "poprawia atrakcyjność", "pomaga wartość",  
        "opracowana dla", "bazująca na", "rozproszona", "sieciowa", "skoncentrowana na", "podniesiona na wyższy  
        poziom", "skierowanej", "udostępniona"};  
  
        String[] listaSlow3 = {"procesu", "punktu wpisywania", "rozwiązania", "strategii", "paradygmatu",  
        "portalu", "witryny", "wersji", "misji"};  
  
        // określenie, ile jest słów w każdej z list  
        int lista1Dlugosc = listaSlow1.length;  
        int lista2Dlugosc = listaSlow2.length;  
        int lista3Dlugosc = listaSlow3.length;  
  
        // generacja trzech losowych słów (lub zwrotów)  
        int rnd1 = (int) (Math.random() * lista1Dlugosc);  
        int rnd2 = (int) (Math.random() * lista2Dlugosc);  
        int rnd3 = (int) (Math.random() * lista3Dlugosc);  
  
        // stworzenie zdanie  
        String zdanie = listaSlow1[rnd1] + " " + listaSlow2[rnd2] + " " + listaSlow3[rnd3];  
  
        // a teraz je zwracamy  
        return ("To jest to, czego nam trzeba: " + zdanie);  
    }  
}
```

Enterprise JavaBeans

— RMI na środkach dopingujących

Technologia RMI doskonale nadaje się do tworzenia i uruchamiania zdalnych usług. Jednak aplikacji takich jak Amazon lub eBey nie dałoby się uruchomić, korzystając wyłącznie z RMI. W przypadku tworzenia bardzo dużych, śmiertelnie poważnych aplikacji biznesowych będziesz potrzebował czegoś więcej. Będziesz potrzebował rozwiązań, które jest w stanie obsługiwać transakcje, doskonale radzić sobie ze współbieżną realizacją zadań (na przykład milionem osób, które jednocześnie chcą kupić organiczną karmę dla psów), zapewnia odpowiedni poziom bezpieczeństwa (nie każdy powinien mieć dostęp do bazy danych rozliczeń finansowych) i udostępnia mechanizmy zarządzania danymi. W takich przypadkach będziesz potrzebował serwera aplikacji korporacyjnych.

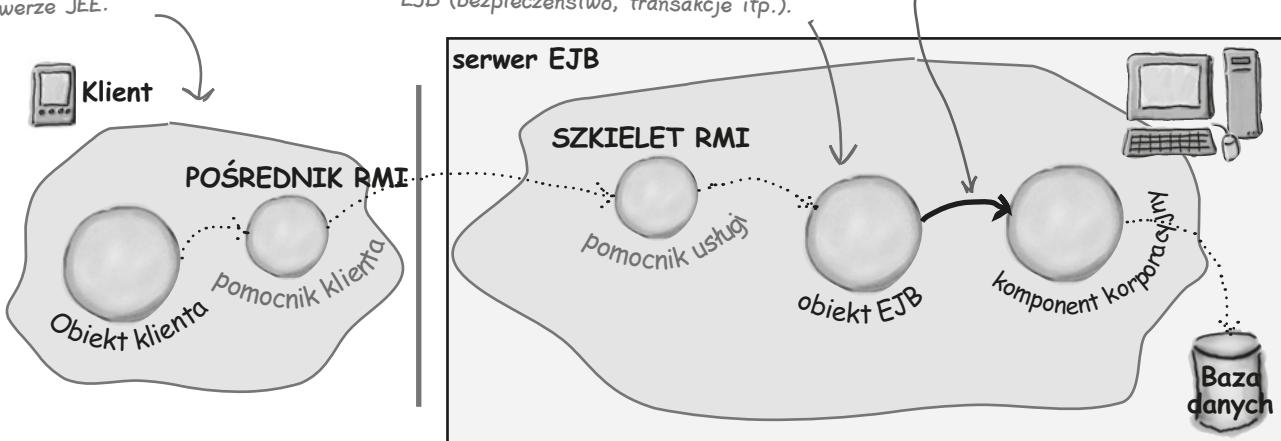
W świecie Javy oznacza to, że będziesz potrzebował serwera *Java Enterprise Edition (JEE)*. Serwer taki zawiera jednocześnie serwer WWW, jak również serwer komponentów *Enterprise JavaBeans (EJB)*, dzięki czemu pozwala na wdrażanie aplikacji, wykorzystujących jednocześnie serwlety, jak i EJB. Zagadnienia związane z tworzeniem i stosowaniem komponentów EJB, podobnie zresztą jak problematyka serwletów, znacznie wykracza poza ramy tematyczne niniejszej książki. Nie mamy możliwości, aby przedstawić choćby najprostszy przykład użycia EJB, zamieścimy jednak krótki opis działania tej technologii. Jeśli interesuje Cię doskonaly, szczegółowy opis tych zagadnień, to polecamy książkę *Enterprise JavaBeans* autorstwa Richarda Monsona-Heafela.

Serwer EJB udostępnia kilka usług, z których nie można korzystać w przypadku stosowania zwyczajnej technologii RMI. Można do nich zaliczyć: transakcje, bezpieczeństwo, współbieżność, zarządzanie bazami danych oraz obsługę połączeń sieciowych.

Serwer EJB stanowi „warstwę pośrednią”, przez którą są przekazywane wywołania RMI, i zapewnia możliwość korzystania ze wszystkich z wymienionych wcześniej usług.

Obiekt komponentu jest zabezpieczony przed bezpośrednim dostępem ze strony klientów. Jedynie serwer jest w stanie komunikować się z tymi obiektemi. Dzięki temu serwer zyskuje nowe możliwości, na przykład może stwierdzić: „Chwila! Mechanizmy zabezpieczeń nie pozwalały temu klientowi na wywołanie tej metody...”. Niemal wszystko, za co płacisz, nabywając serwer EJB, dzieje się w TYM miejscu, tu bowiem serwer wkraca do akcji!

Klient może być CZYMKOLWIEK, jednak zazwyczaj klientami EJB są serwlety działające na tym samym serwerze JEE.



To jedynie niewielka część całego rysunku prezentującego EJB.

I na samym końcu przedstawiamy... małego dżina Jini

Uwielbiamy Jini. Uważamy, że Jini jest jednym z najlepszych rozwiązań, jakie daje Java. Jeśli EJB można porównać z RMI na środkach dopingujących (i całą masą menadżerów), to Jini trzeba by uznać za RMI ze skrzydłami. Najdoskonalszą rozkosc w świecie Javy. Nie jesteśmy w stanie przedstawić w tym rozdziale wyczerpującego omówienia technologii Jini, podobnie jak nie byliśmy w stanie omówić EJB. Jednak jeśli znasz RMI, to będziesz już w trzech czwartych drogi do celu. Przynajmniej jeśli chodzi o zagadnienia związane z technologią. Jeśli chodzi o *nastrój*, to czas go radykalnie poprawić. Czas wzbić się w *chmury*.

Jini wykorzystuje RMI (choć można także stosować inne protokoły), jednak dodaje kilka kluczowych możliwości, takich jak:

Odkrywanie adaptacyjne.

Sieci „samoaktualizujące”.

Pamiętaj, że w przypadku stosowania RMI klient musi znać nazwę oraz lokalizację zdalnej usługi. Umieszczany w kliencie kod wyszukujący usługę zawiera adres IP lub nazwę hosta tej usługi (ponieważ na tym komputerze działa rejestr RMI) oraz logiczną nazwę, pod jaką usługa ta została zarejestrowana.

Z kolei w przypadku stosowania technologii Jini klient musi dysponować tylko jedną informacją — *musi znać interfejs implementowany przez usługę!* I to wszystko.

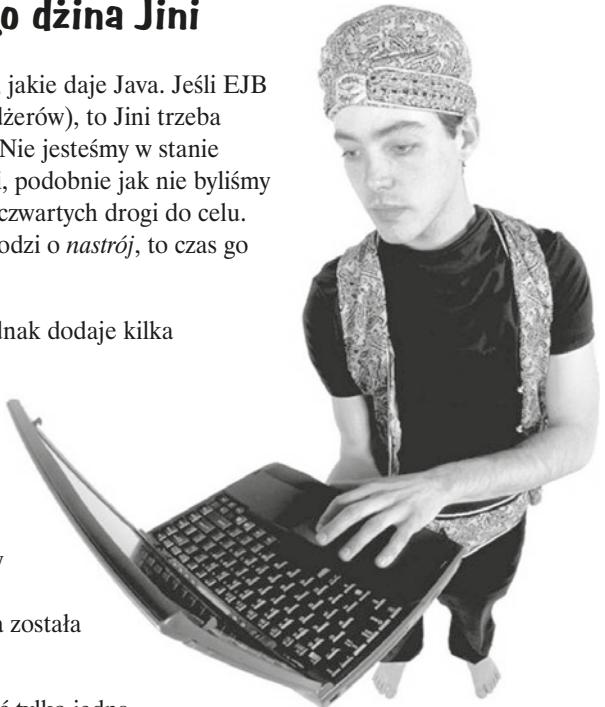
Zatem w jaki sposób można znaleźć usługę? Cała sztuczka jest związana z usługami lokalizacji, jakimi dysponuje Jini. Usługi te są znacznie bardziej elastyczne i mają znacznie większe możliwości niż rejestr RMI. Przede wszystkim usługi lokalizacji Jini ogłaszały się w sieci, a co najważniejsze — robią to *automatycznie*. Kiedy usługa zostaje uruchomiona, rozsyła w sieci komunikat (wykorzystując mechanizmy rozsyłania grupowego IP): „Oto jestem, jeśli kogokolwiek to interesuje”.

Ale to nie wszystko. Założymy, że Ty (czyli klient) podłączyłeś się do sieci po tym, jak usługa ogłosiła swoją obecność. W takim przypadku możesz rozesłać w całej sieci komunikat z pytaniem: „Czy są tam jakieś usługi lokalizacji?”.

Jednak tak naprawdę nie interesują Cię same usługi lokalizacji, lecz usługi, które zostały w nich *zarejestrowane*. Mogą to być „rzeczy” podobne do zdalnych usług RMI, wszelkiego typu inne obiekty posiadające możliwość serializacji, a nawet urządzenia, takie jak drukarki, kamery czy też eksprezy do kawy.

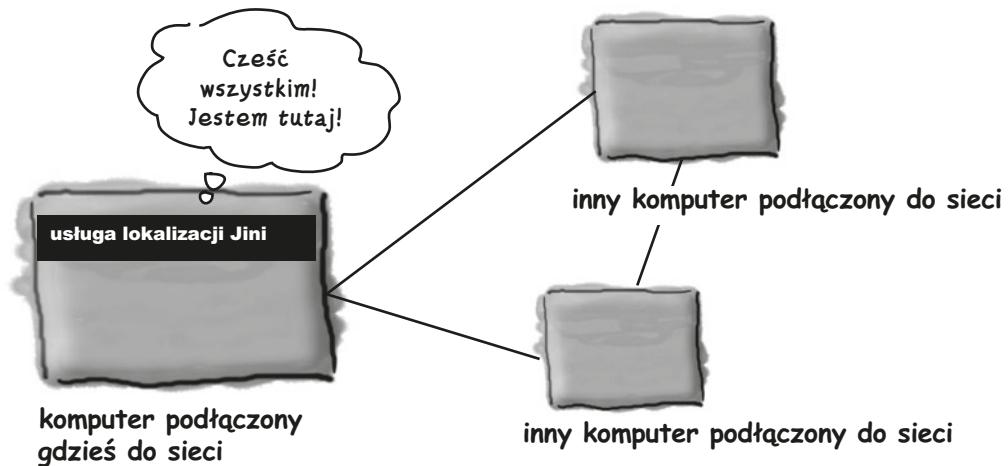
I w końcu dochodzimy do tego, co w Jini jest najlepszego - otóż po udostępnieniu usługa dynamicznie wykryje wszystkie usługi lokalizacji dostępne w sieci (i *zarejestruje* się w nich). Podczas rejestracji w usłudze lokalizacji usługa przesyła do niej serializowany obiekt. Obiektem tym może być pośrednik pozwalający na korzystanie ze zdalnej usługi RMI, sterownik jakiegoś urządzenia sieciowego, a nawet sama usługa, która po skopiowaniu zostanie uruchomiona na lokalnym komputerze. Co więcej, usługi nie są rejestrowane na podstawie *nazwy*, lecz implementowanego *interfejsu*.

Kiedy już zdobędziesz (Ty — czyli klient) odwołanie do usługi lokalizacji, możesz ją zapytać: „Hej, czy dysponujesz czymkolwiek, co implementuje KalkulatorNaukowy?”. W tym momencie usługa lokalizacji sprawdzi swoją listę zarejestrowanych interfejsów i, zakładając, że znajdzie to, czego szukasz, odpowie: „Tak, faktycznie mam coś, co implementuje ten interfejs. Oto serializowany obiekt, który zarejestrowała usługa KalkulatorNaukowy”.

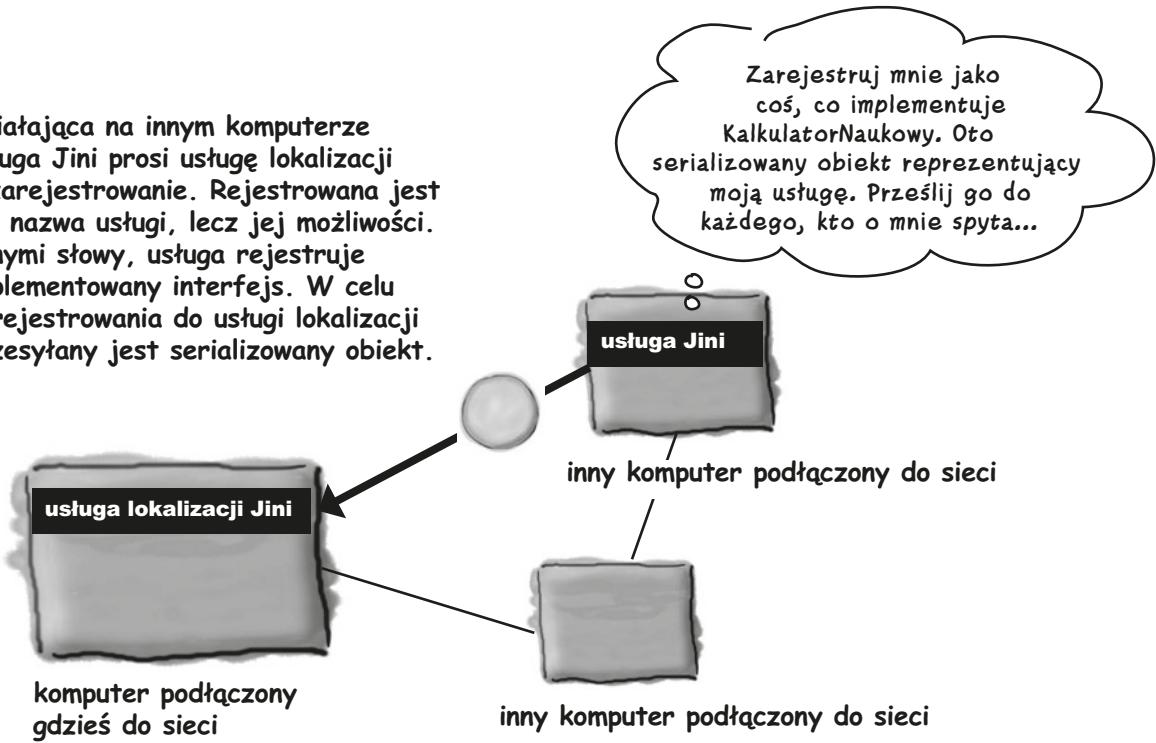


Odkrywanie adaptacyjne w akcji

- ① Gdzieś w sieci zostaje uruchomiona usługa lokalizacji Jini, która ogłasza swoją obecność przy wykorzystaniu rozsyłania grupowego IP.

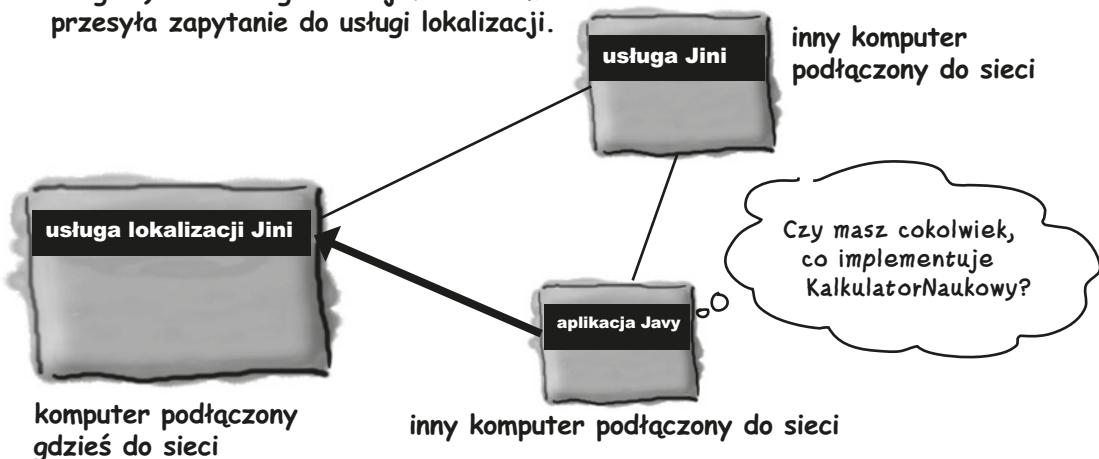


- ② Działająca na innym komputerze usługa Jini prosi usługę lokalizacji o zarejestrowanie. Rejestrowana jest nie nazwa usługi, lecz jej możliwości. Innymi słowy, usługa rejestruje implementowany interfejs. W celu zarejestrowania do usługi lokalizacji przesyłany jest serializowany obiekt.

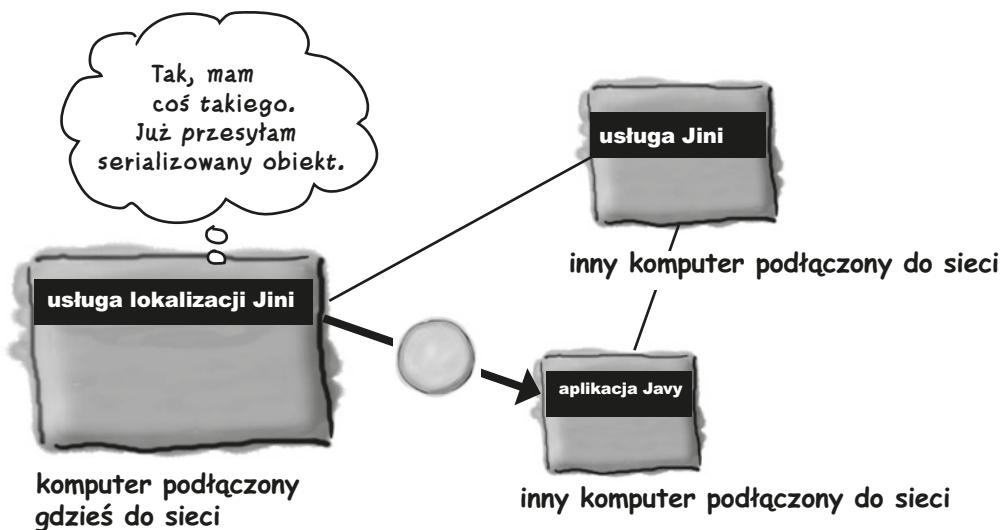


Odkrywanie adaptacyjne w akcji, ciąg dalszy...

- 3 Klient chciałby znaleźć coś, co implementuje interfejs KalkulatorNaukowy. Nie ma przy tym najmniejszego pojęcia, gdzie (jeśli w ogóle) coś takiego istnieje. A zatem przesyła zapytanie do usługi lokalizacji.

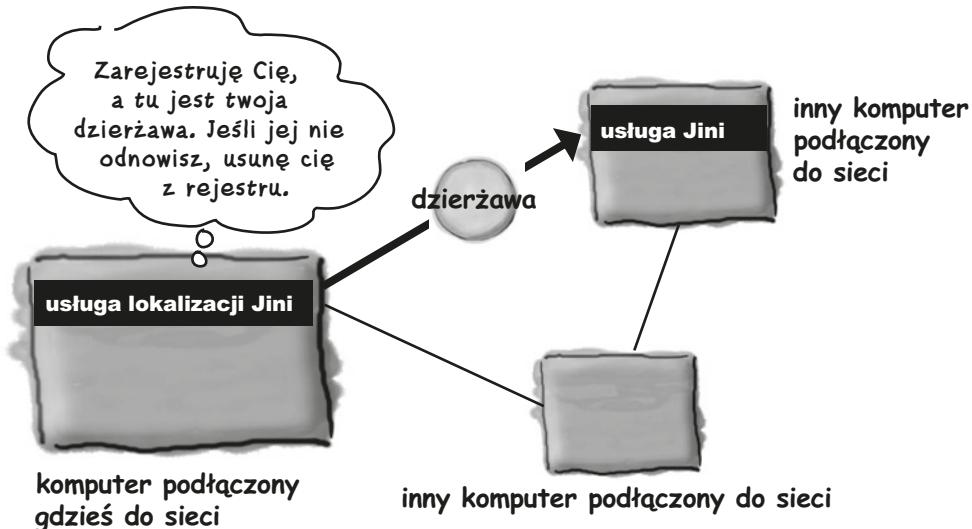


- 4 Usługa lokalizacji odpowiada, gdyż faktycznie został w niej zarejestrowany interfejs KalkulatorNaukowy.

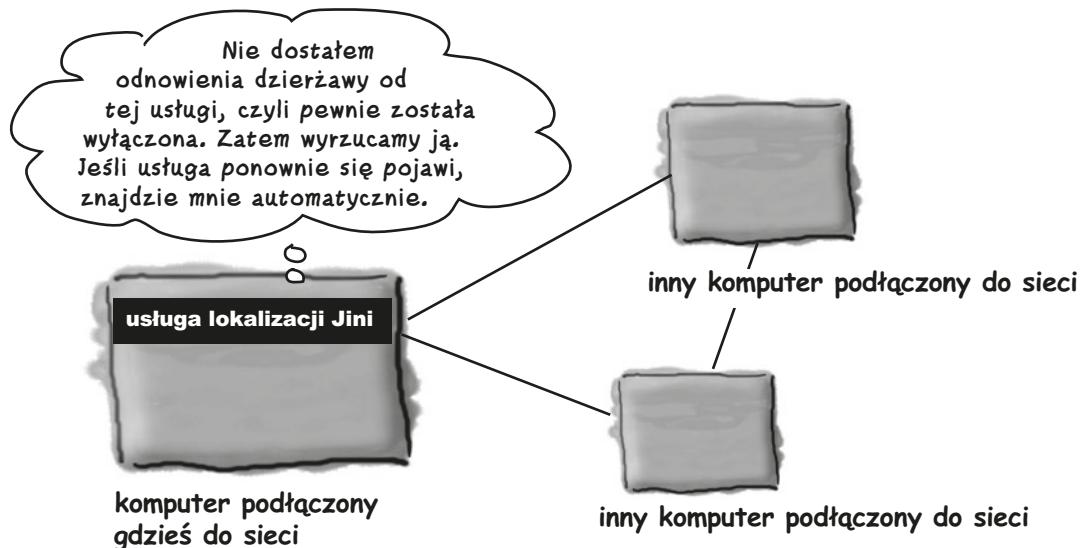


Sieci „samoaktualizujące” w działaniu

- 1 Usługa Jini poprosiła o rejestrację w usłudze lokalizacji. Usługa lokalizacji w odpowiedzi przekazuje jej „dzierżawę”. Dzierżawę tę trzeba odnawiać, gdyż w przeciwnym razie usługa lokalizacji uzna, że zarejestrowana usługa przestała być dostępna w sieci. Usługa lokalizacji zawsze chce przedstawić aktualny stan usług dostępnych w sieci.



- 2 Usługa przestaje być dostępna (bo ktoś ją wyłączył), a zatem nie odnawia dzierżawy otrzymanej od usługi lokalizacji. Usługa lokalizacji usuwa naszą usługę ze swego rejestru.



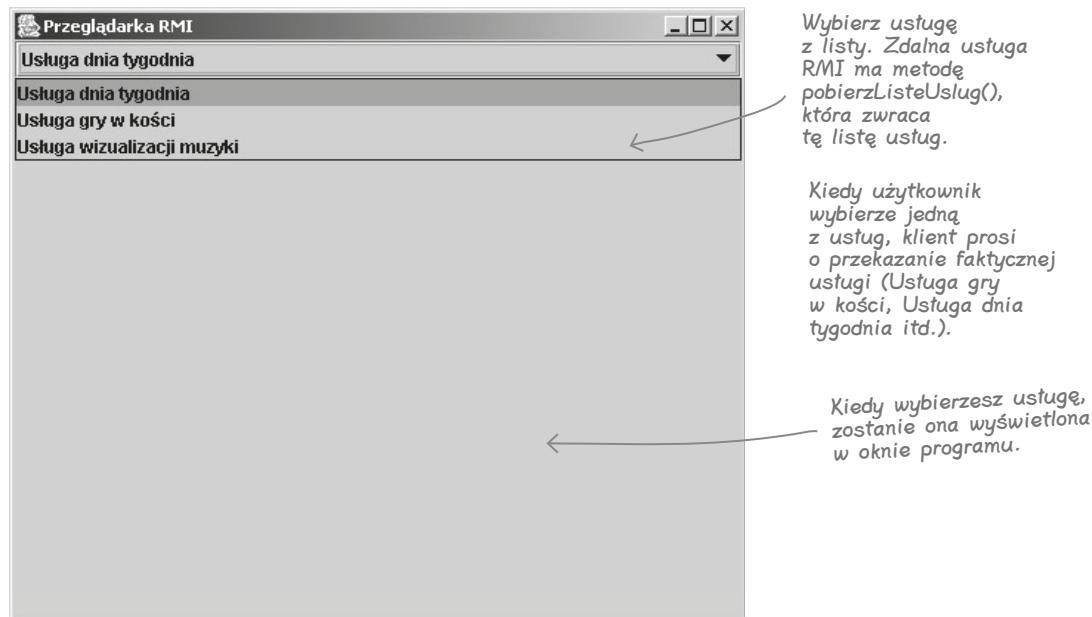
Ostatni projekt — przeglądarka usług uniwersalnych

Mamy zamiar stworzyć program, który co prawda nie będzie korzystać z Jini, lecz bardzo łatwo można by w nim tę technologię zastosować. Program da Ci przedsmak możliwości, jakie zapewnia Jini, lecz będzie korzystał ze zwykłej technologii RMI. W rzeczywistości podstawowa różnica pomiędzy przedstawioną tu aplikacją a analogiczną aplikacją Jini polega na sposobie odkrywania usług. Zamiast usługi lokalizacji Jini, która automatycznie się ogłasza i jest obecna w każdym miejscu sieci, używamy rejestru RMI. Jak wiadomo rejestr RMI musi działać na tym samym komputerze, na którym działa zdalna usługa, a co więcej, nie rozgłasza automatycznie swojej obecności.

Poza tym nasza usługa nie zarejestruje się automatycznie w usłudze lokalizacji Jini — zamiast tego *sam* musimy ją zarejestrować w rejestrze RMI (wykorzystując w tym celu metodę `Naming.rebind()`).

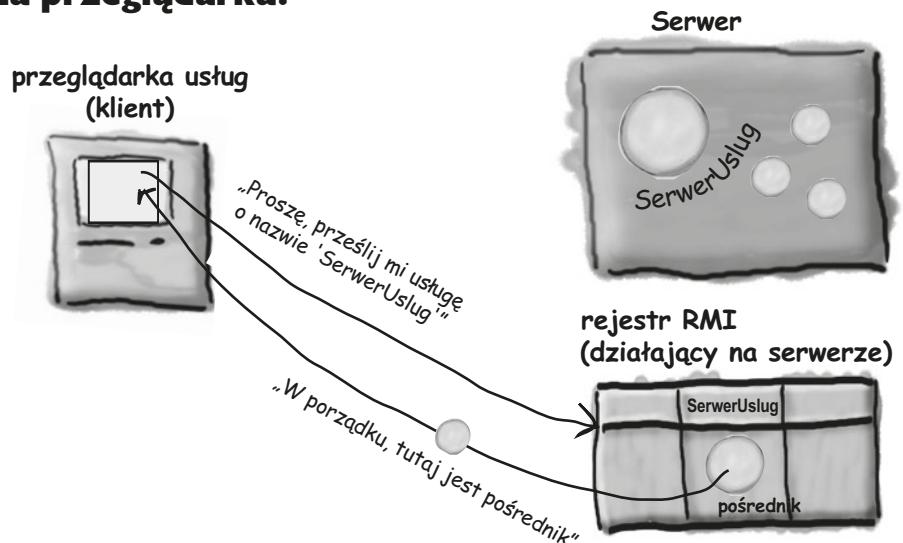
Jednak kiedy klient odnajdzie usługę w rejestrze RMI, dalszy sposób działania naszej aplikacji jest niemal identyczny jak w przypadku aplikacji Jini. (Podstawowa różnica polega na braku *dzierżawy*, która pozwoliłaby na automatyczną aktualizację stanu sieci w przypadku, gdyby jedna z usług przestała być dostępna).

Uniwersalna przeglądarka usług przypomina nieco wyspecjalizowaną przeglądarkę WWW z tą różnicą, iż zamiast dokumentów HTML nasza przeglądarka wyświetla interaktywne interfejsy użytkownika, które nazywamy *usługami uniwersalnymi*.

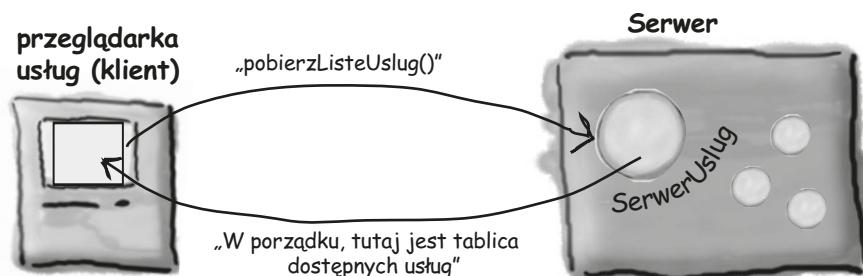


A oto jak działa nasza przeglądarka:

- ① Klient zostaje uruchomiony i przegląda rejestr RMI w poszukiwaniu usługi o nazwie „SerwerUslug”, po czym pobiera pośrednika.



- ② Klient wywołuje metodę pobierzListeUslug() pośrednika. Metoda zwraca tablicę dostępnych usług.

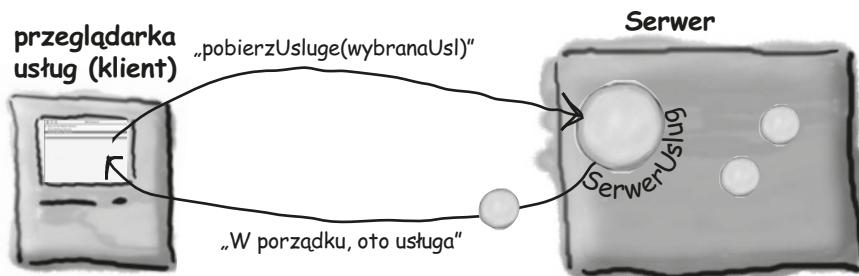


- ③ Klient wyświetla listę dostępnych usług.



A oto jak działa nasza przeglądarka, ciąg dalszy...

- 4 Użytkownik wybiera jedną z usług z listy, a zatem klient wywołuje metodę pobierzUsluge() usługi zdalnej. Zdalna usługa zwraca serializowany obiekt będący faktyczną usługą, która zostanie uruchomiona w przeglądarce.



- 5 Klient wywołuje metodę pobierzPanelGUI() otrzymanego wcześniej obiektu. Graficzny interfejs użytkownika definiowany przez usługę jest wyświetlany w przeglądarce, a użytkownik może z niego korzystać (lokalnie). W tym momencie nie potrzebujemy już zdalnej usługi, chyba że użytkownik zdecyduje się wybrać inną usługę.

przeglądarka
usług (klient)



Klasy i interfejsy

1 Interfejs SerwerUslug implementuje Remote.

Zwyczajny zdalny interfejs RMI wykorzystywany przez zdальną usługę (zdalna usługa ma metody służące do pobierania listy usług oraz wybranej usługi).



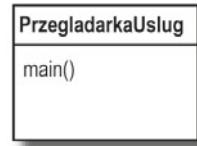
2 Klasa SerwerUslugImpl implementuje SerwerUslug.

Właściwa zdalna usługa RMI (rozszerzająca klasę UnicastRemoteObject). Jej zadaniem jest stworzenie i przechowywanie usług (obiektów, które będą przekazywane do klienta) oraz zarejestrowanie samego serwera usług (obiektu SerwerUslugImpl) w rejestrze RMI.



3 Klasa PrzegladarkaUslug.

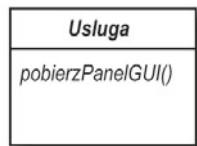
Część „klienta” naszej aplikacji. Klasa tworzy bardzo prosty graficzny interfejs użytkownika, przeszukuje rejestr RMI i pobiera z niego pośrednika SerwerUslug, a następnie używa go do wywołania zdalnej metody zwracającej listę dostępnych usług; lista ta zostaje następnie wyświetlona w oknie programu.



4 Interfejs Usluga.

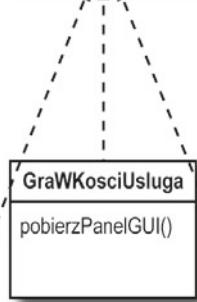
To kluczowy element całej aplikacji. Ten bardzo prosty interfejs zawiera tylko jedną metodę – pobierzPanelGUI(). Każdy klient, który będzie przesyłany do klienta musi implementować ten interfejs. To właśnie on sprawia, że nasze usługi są uniwersalne! Dzięki temu, że każda usługa musi implementować ten interfejs, klient będzie mógł jej użyć, nawet nie znając nazwy klasy (lub klas) definiującej tę usługę. Klient wie tylko tyle, iż każda przekazywana do niego usługa implementuje interfejs Usluga, a zatem **MUSI** udostępniać metodę pobierzPanelGUI().

W wyniku wywołania metody pobierzUsluge(wybranaUsl) obiektu pośrednika klient uzyskuje serializowany obiekt, któremu oznajmia: „Słuchaj, nie interesuje mnie, kim albo czym jesteś, ale wiem, że implementujesz interfejs Usluga, czyli że mogę wywołać twoją metodę pobierzPanelGUI(). A ponieważ ta metoda zwraca obiekt JPanel, zatem mogę go wyświetlić w oknie mojej przeglądarki i zacząć go używać.”



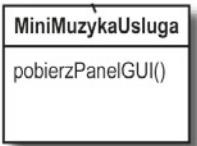
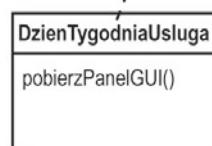
5 Klasa GraWKosciUsluga implementuje Usluga.

Masz kości do gry? Jeśli nie masz, ale ich potrzebujesz, to możesz wykorzystać tę usługę, aby „wyrzuciła” za Ciebie od jednej do pięciu wirtualnych kości.



6 Klasa MiniMuzykaUsluga implementuje Usluga.

Czy pamiętasz nasz wspaniały program tworzący „muzyczne wideo” przedstawiony w pierwszej części „Kodu od kuchni” poświęconej graficznemu interfejsowi użytkownika? Zamieniliśmy go w usługę, dzięki czemu będziesz mógł odtwarzać go w kótko tak długo, aż Twój kolega z pokoju w końcu się zdenerwuje i wyjdzie.



7 Klasa DzienTygodniaUsluga implementuje Usluga.

Czy urodziłeś się w piątek? Podaj datę urodzin i sprawdź.

Kod uniwersalnej usługi

Interfejs SerwerUslug (zdalny interfejs)

```
import java.rmi.*;  
  
public interface SerwerUslug extends Remote {  
    Object[] pobierzListeUslug() throws RemoteException;  
    Usluga pobierzUsluge(Object serviceKey) throws RemoteException;  
}
```

Zwykły zdalny interfejs RMI definiujący dwie metody, które będzie udostępniać zdalna usługa.

Interfejs Usluga (implementowany przez usługi GUI)

```
import javax.swing.*;  
import java.io.*;  
  
public interface Usluga extends Serializable {  
    public JPanel pobierzPanelGUI();  
}
```

Zwykły stary (czyli niezdalny) interfejs definiujący jedną metodę, którą muszą mieć wszystkie uniwersalne usługi — metodę pobierzPanelGUI(). Interfejs ten rozszerza interfejs Serializable, dzięki czemu każda klasa implementująca interfejs Usluga automatycznie będzie posiadać możliwość serializacji.

To rozwiązanie jest konieczne, gdyż usługa jest przekazywana siecią z serwera do klienta w wyniku wywołania metody pobierzUsluge() zdalnej usługi SerwerUslug.

Klasa SerwerUslugImpl (klasa implementująca zdalny interfejs)

```

import java.rmi.*;
import java.util.*;
import java.rmi.server.*;

public class SerwerUslugImpl extends UnicastRemoteObject implements SerwerUslug {

    HashMap listaUslug;

    public SerwerUslugImpl() throws RemoteException {
        konfigurujUslugi();
    }

    private void konfigurujUslugi() {
        listaUslug = new HashMap();
        listaUslug.put("Usługa gry w kości", new GraWKosciUsluga());
        listaUslug.put("Usługa dnia tygodnia", new DzienTygodniaUsluga());
        listaUslug.put("Usługa wizualizacji muzyki", new MiniMuzykaUsluga());
    }

    public Object[] pobierzListeUslug() {
        System.out.println("zdalne");
        return listaUslug.keySet().toArray();
    }

    public Usluga pobierzUslugę(Object kluczUsl) throws RemoteException {
        Usluga usluga = (Usluga) listaUslug.get(kluczUsl);
        return usluga;
    }

    public static void main (String[] args) {
        try {
            Naming.rebind("SerwerUslug", new SerwerUslugImpl());
        } catch(Exception ex) { }
        System.out.println("Zdalna usługa uruchomiona");
    }
}

```

Normalna implementacja RMI.

Uslugi będą przechowywane w kolekcji `HashMap`. W tym przypadku w kolekcji umieszczany jest nie jeden, lecz DWA obiekty — obiekt klucza (na przykład obiekt `String`) oraz obiekt wartości (catkowicie dowolny). (Więcej informacji na temat klasy `HashMap` można znaleźć w dodatku B).

W momencie wywołania konstruktora zostają zainicjowane nasze uniwersalne usługi (`GraWKosciUsluga`, `MiniMuzykaUsluga` itd.).

Tworzymy usługi (faktyczne obiekty usług) i zapisujemy je w kolekcji `HashMap`, kojarząc z odpowiednimi tańcuchami znaków (petniącymi funkcję „kluczy”).

Klient wywołuje tę metodę w celu pobrania listy usług, które można wyświetlić w przeglądarce (dzięki czemu użytkownik będzie mógł wybrać jedną z nich). Metoda zwraca tablicę typu `Object` (choć faktycznie są w niej przechowywane obiekty `String`), która stanowi pobrana z kolekcji `HashMap` tablica KLUCZY. Konkretny obiekt uniwersalnej usługi zostanie przestany dopiero w momencie, gdy użytkownik zażąda go, wywołując metodę `pobierzUslugę()`.

Kod klasy PrzegladarkaUslug

Klasa PrzegladarkaUslug (klient)

```
import java.awt.*;
import javax.swing.*;
import java.rmi.*;
import java.awt.event.*;

public class PrzegladarkaUslug {

    JPanel panelGlowny;
    JComboBox listaUslug;
    SerwerUslug serwer;

    public void tworzGUI() {
        JFrame ramka = new JFrame("Przeglądarka RMI");
        panelGlowny = new JPanel();
        ramka.getContentPane().add(BorderLayout.CENTER, panelGlowny);

        Object[] uslugi = pobierzListeUslug(); ←
        listaUslug = new JComboBox(uslugi);
        ramka.getContentPane().add(BorderLayout.NORTH, listaUslug);

        listaUslug.addActionListener(new ListaUslugListener());

        ramka.setSize(500,500);
        ramka.setVisible(true);
    }

    void wczytajUsluge(Object wybranaUsl) {
        try {
            Usluga usl = serwer.pobierzUsluge(wybranaUsl);

            panelGlowny.removeAll();
            panelGlowny.add(usl.pobierzPanelGUI());
            panelGlowny.validate();
            panelGlowny.repaint();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Ta metoda przegląda rejestr RMI, pobiera pośrednika i wywołuje metodę pobierzListeUslug(). (Kod tej metody zostanie przedstawiony na następnej stronie).

Dodajemy listę usług (tablicę obiektów Object) do komponentu JComboBox (rozwijanej listy). Komponent sam potrafi pobrać z tej tablicyłańcuchy znaków, które zostaną wyświetlane na liście.

To właściwie w tym miejscu dodajemy usługę wybraną przez użytkownika do graficznego interfejsu przeglądarki (ta metoda jest wywoływana przez odbiorcę zdarzeń komponentu JComboBox). W pierwszej kolejności wywołujemy metodę pobierzUsluge() zdalnego serwera (używając przy tym pośrednika SerwerUslug), przekazując w jej wywołaniułańcuch znaków wyświetlony na liście (jest to ten samłańcuch znaków, który na samym początku przestanął nam serwer w wyniku wywołania metody pobierzListeUslug()). W odpowiedzi serwer przesyła nam serializowany obiekt usługi, który (dzięki RMI) zostaje automatycznie poddany deserializacji. Nie pozostaje nam zatem nic innego jak wywołać metodę pobierzPanelGUI() usługi, a zwrócony wynik (obiekt JPanel) wyświetlić w panelu głównym przeglądarki.

```

Object[] pobierzListeUslug() {

    Object obj = null;
    Object[] uslugi = null;

    try {
        obj = Naming.lookup("rmi://127.0.0.1/SerwerUslug");
    }
    catch(Exception ex) {
        ex.printStackTrace();
    }
    serwer = (SerwerUslug) obj;

    try {
        uslugi = serwer.pobierzListeUslug();
    } catch(Exception ex) {
        ex.printStackTrace();
    }
    return uslugi;
}

class ListaUslugListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
        // pobranie wybranej usługi
        Object wybor = listaUslug.getSelectedItem();
        wczytajUsluge(wybor);
    }
}

public static void main(String[] args) {
    new PrzegladarkaUslug().tworzGUI();
}
}

```

Przeszukujemy rejestr RMI i pobieramy pośrednika.

Rzutujemy pośrednika na typ zdalnego interfejsu, dzięki czemu będziemy mogli wywołać metodę pobierzListeUslug().

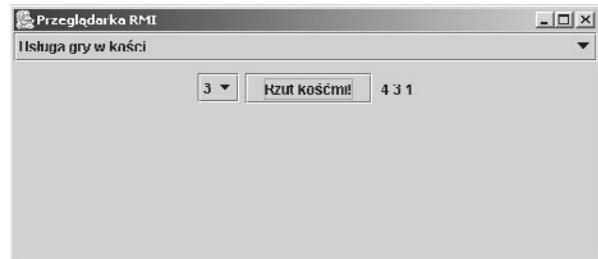
Metoda pobierzListeUslug() zwraca tablicę obiektów typu Object, którą następnie wyświetlaną w komponentie JComboBox, dając użytkownikowi możliwość wybrania jednej z dostępnych usług.

Jeśli wykonujemy ten wiersz kodu, oznacza to, że użytkownik wybrał jedną z usług z listy JComboBox, a zatem określamy, jaka opcja została wybrana i wczytujemy odpowiednią usługę. (Patrz metoda wczytajUsluge() przedstawiona na poprzedniej stronie; metoda ta prosi serwer o przestanie usługi skojarzonej z podanym ciągkiem znaków).

Kod klasy GraWKosciUsluga

Klasa GraWKosciUsluga (usługa uniwersalna, klasa implementuje interfejs Usluga)

```
import javax.swing.*;  
import java.awt.event.*;  
  
public class GraWKosciUsluga implements Usluga {  
  
    JLabel etykieta;  
    JComboBox iloscKosci;  
  
    public JPanel pobierzPanelGUI() {  
        JPanel panel = new JPanel();  
        JButton przycisk = new JButton("Rzut kośćmi!");  
        String[] opcje = {"1", "2", "3", "4", "5"};  
        iloscKosci = new JComboBox(opcje);  
        etykieta = new JLabel("wylosowane cyfry");  
        przycisk.addActionListener(new RzutKoscmiListener());  
        panel.add(iloscKosci);  
        panel.add(przycisk);  
        panel.add(etykieta);  
        return panel;  
    }  
  
    public class RzutKoscmiListener implements ActionListener {  
        public void actionPerformed(ActionEvent ev) {  
            // rzucamy kości  
            String wynikiRzutu = "";  
            String wybor = (String) iloscKosci.getSelectedItem();  
            int iloscUzywanychKosci = Integer.parseInt(wybor);  
            for (int i = 0; i < iloscUzywanychKosci; i++) {  
                int r = (int) ((Math.random() * 6) + 1);  
                wynikiRzutu += (" " + r);  
            }  
            etykieta.setText(wynikiRzutu);  
        }  
    }  
}
```

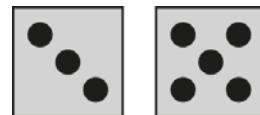


Oto najważniejsza metoda! Metoda interfejsu Usluga — to właśnie ona zastanie wywołana, kiedy klient wybierze i wczyta tę usługę. W metodzie pobierzPanelGUI() można wykonywać dowolne operacje pod warunkiem, że w efekcie zostanie zwrócony obiekt JPanel. W tym przypadku metoda pobierzPanelGUI() tworzy interfejs graficzny pozwalający na „rzucanie wirtualnymi kościemi”.



Zaostrz ołówek

Pomyśl, w jaki sposób można by poprawić klasę GraWKosciUsluga. Oto pewna sugestia: bazując na informacjach zdobytych w rozdziałach poświęconych tworzeniu graficznego interfejsu użytkownika, zmień klasę w taki sposób, aby „kości” były prezentowane graficznie. Wykorzystaj w tym celu prostokąty, a wewnątrz każdego z nich narysuj kółka, których ilość będzie odpowiadać liczbie oczek wylosowanej na konkretnej „kości”.



Klasa MiniMuzykaUsluga (usługa uniwersalna, klasa implementuje interfejs Usluga)

```

import javax.sound.midi.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MiniMuzykaUsluga implements Usluga {

    MojPanelRysunkowy mojPanel;

    public JPanel pobierzPanelGUI() { ←
        JPanel panelGlowny = new JPanel();
        mojPanel = new MojPanelRysunkowy();
        JButton przyciskZagraj = new JButton("Zagraj!");
        przyciskZagraj.addActionListener(new ZagrajListener());
        panelGlowny.add(mojPanel);
        panelGlowny.add(przyciskZagraj);
        return panelGlowny;
    }

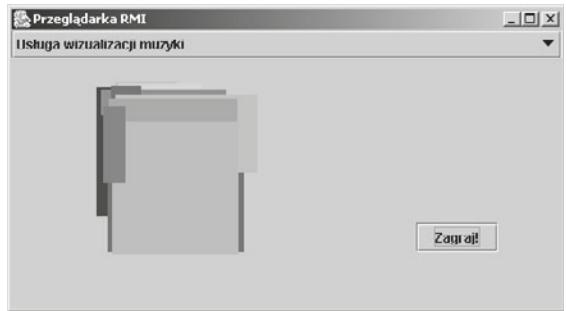
    public class ZagrajListener implements ActionListener {
        public void actionPerformed(ActionEvent ev) {
            try {
                Sequencer sekvenser = MidiSystem.getSequencer();
                sekvenser.open();

                sekvenser.addControllerEventListener(mojPanel, new int[] {127});
                Sequence sekwencja = new Sequence(Sequence.PPQ, 4);
                Track track = sekwencja.createTrack();

                for (int i = 0; i < 100; i+= 4) {
                    int rNum = (int) ((Math.random() * 50) + 1);
                    if (rNum < 38) { // dalsze czynności wykonujemy, jeśli num <38 (75% czasu)
                        track.add(makeEvent(144,1,rNum,100,i));
                        track.add(makeEvent(176,1,127,0,i));
                        track.add(makeEvent(128,1,rNum,100,i + 2));
                    }
                } // koniec pętli

                sekvenser.setSequence(sekwencja);
                sekvenser.start();
                sekvenser.setTempoInBPM(220);
            } catch (Exception ex) {ex.printStackTrace();}
        } // koniec metody actionPerformed
    } // koniec klasy wewnętrznej
}

```



Metoda usługi! Jedynym jej zadaniem jest wyświetlenie przycisku i obsługa rysowania (czyli wyświetlanie prostokątów w panelu).

Ciąg pozostała część kodu to metody muzyczne zaczerpnięte z „Kodu od kuchni” z rozdziału 12., dlatego też nie będziemy szczegółowo opisywać tego kodu.

Kod klasy MiniMuzykaUsluga

Klasa MiniMuzykaUsluga, ciąg dalszy...

```
public MidiEvent twozZdarzenie(int plc, int kanal, int jeden, int dwa, int takt) {  
    MidiEvent zdarzenie = null;  
    try {  
        ShortMessage a = new ShortMessage();  
        a.setMessage(plc, kanal, jeden, dwa);  
        zdarzenie = new MidiEvent(a, takt);  
    }catch(Exception e) {}  
    return zdarzenie;  
}  
}
```

```
class MojPanelRysunkowy extends JPanel implements ControllerEventListener {  
    // rysujemy tylko wtedy, jeśli odebraliśmy zdarzenie  
    boolean kmk = false;
```

```
    public void controlChange(ShortMessage event) {  
        kmk = true;  
        repaint();  
    }
```

```
    public Dimension getPreferredSize() {  
        return new Dimension(300,300);  
    }
```

```
    public void paintComponent(Graphics g) {  
        if (kmk) {  
  
            Graphics2D g2 = (Graphics2D) g;  
  
            int r = (int) (Math.random() * 250);  
            int gr = (int) (Math.random() * 250);  
            int b = (int) (Math.random() * 250);  
  
            g.setColor(new Color(r,gr,b));  
  
            int wys = (int) ((Math.random() * 120) + 10);  
            int szer = (int) ((Math.random() * 120) + 10);  
  
            int x = (int) ((Math.random() * 40) + 10);  
            int y = (int) ((Math.random() * 40) + 10);  
  
            g.fillRect(x,y,wys, szer);  
            kmk = false;  
        } // koniec if  
    } // koniec metody  
} // koniec klasy wewnętrznej  
} // koniec klasy
```

W kodzie przedstawionym na tej stronie nie ma niczego nowego. Widziałeś go już w „Kodzie od kuchni” zamieszczonym w rozdziałach dotyczących graficznego interfejsu użytkownika. Jeśli chcesz poćwiczyć, to spróbuj sam opisać ten kod, a następnie porównaj swoje notatki z tymi, które zostały zamieszczone w rozdziale 12.

Klasa DzienTygodniaUsluga (usługa uniwersalna, klasa implementuje interfejs Usluga)

```

import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import java.text.*;

public class DzienTygodniaUsluga implements Usluga {

    JLabel etykietaWyniku;
    JComboBox miesiac;
    JTextField dzien;
    JTextField rok;

    public JPanel pobierzPanelGUI() {
        JPanel panel = new JPanel();
        JButton przycisk = new JButton("Wyznacz!");
        przycisk.addActionListener(new WyznaczListener());
        etykietaWyniku = new JLabel("tu jest wyświetlana data");
        DateFormatSymbols dateStuff = new DateFormatSymbols();
        miesiac = new JComboBox(dateStuff.getMonths());
        dzien = new JTextField(8);
        rok = new JTextField(8);
        JPanel inputPanel = new JPanel(new GridLayout(3,2));
        inputPanel.add(new JLabel("Dzień"));
        inputPanel.add(dzien);
        inputPanel.add(new JLabel("Miesiąc"));
        inputPanel.add(miesiac);
        inputPanel.add(new JLabel("Rok"));
        inputPanel.add(roku);
        panel.add(inputPanel);
        panel.add(przycisk);
        panel.add(etykietaWyniku);
        return panel;
    }

    public class WyznaczListener implements ActionListener {
        public void actionPerformed(ActionEvent ev) {
            int miesiacNum = miesiac.getSelectedIndex();
            int dzienNum = Integer.parseInt(dzien.getText());
            int rokNum = Integer.parseInt(roku.getText());
            Calendar k = Calendar.getInstance();
            k.set(Calendar.MONTH, miesiacNum);
            k.set(Calendar.DAY_OF_MONTH, dzienNum);
            k.set(Calendar.YEAR, rokNum);
            Date date = k.getTime();
            String dzienTygodnia = (new SimpleDateFormat("EEEE")).format(date);
            etykietaWyniku.setText(dzienTygodnia);
        }
    }
}

```

Metoda interfejsu Usluga tworząca graficzny interfejs użytkownika.



Zajrzyj do rozdziału 10., jeśli musisz sobie przypomnieć, jak można formatować liczby i daty. Przedstawiony tu kod jest nieco inny, gdyż wykorzystuje klasę Calendar. Poza tym, klasa SimpleDateFormat pozwala na określenie wzorca wyglądu daty.

Koniec... a przynajmniej coś w tym stylu...



Gratulujemy!

Udało Ci się dotrzeć do końca książki.

Oczywiście wciąż masz przed sobą dwa dodatki.

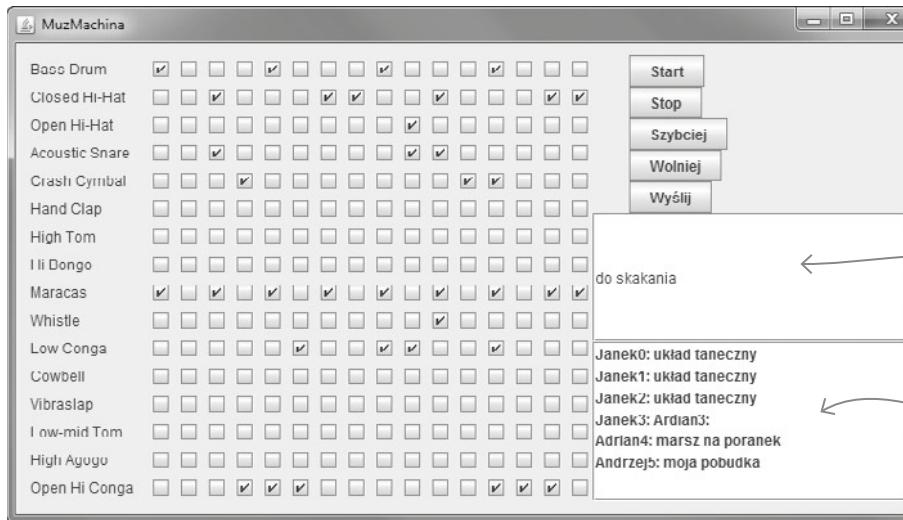
No i indeks.

A poza tym, jest jeszcze witryna WWW.

**Może jednak te gratulacje
były trochę przedwczesne...**

Dodatek A

Ostatnie doprawianie kodu



Kiedy klikniesz przycisk „Wyślij”, Twoja wiadomość zostaje wysyłana do innych uczestników pogawędki wraz z aktualną kompozycją.

Odebrane wiadomości wystane przez innych uczestników pogawędki. Kliknij wiadomość, aby wczytać i odtworzyć utwór przestany wraz z nią.

Ostateczna, pełna wersja naszej muzycznej aplikacji MuzMachina!

Program nawiązuje połączenie z prostym serwerem muzycznym, umożliwiając przesyłanie i odbieranie kompozycji od innych użytkowników.

Ostateczna wersja kodu aplikacji MuzMachine

Ostateczna wersja programu MuzMachine

Znacząca część przedstawionego tu kodu nie różni się niczym do poprzednich wersji, przedstawionych we wcześniejszych rozdziałach. Dlatego też nie będziemy szczegółowo opisywać całego programu. W kodzie pojawiły się jednak pewne nowe dodatki i modyfikacje, związane z:

INTERFEJSEM UŻYTKOWNIKA — dodane zostały dwa komponenty — wielowierszowe pole tekstowe (w zasadzie lista przewijana) służące do wyświetlania odbieranych komunikatów oraz pole tekstowe.

OBSŁUGĄ KOMUNIKACJI SIECIOWEJ — podobnie jak przedstawiony we wcześniejszej części książki program ProstyKlientPogawedek także i ta wersja aplikacji MuzMachine nawiązuje połączenie z serwerem i pobiera strumień wejściowy i wyjściowy.

WĄTKAMI — podobnie jak w programie ProstyKlientPogawedek także i tu tworzymy klasę „czytelnika”, który oczekuje na wiadomości przesyłane z serwera. Jednak w tym przypadku, przesyłane komunikaty nie zawierają wyłącznie tekstu, lecz składają się z dwóch obiektów — łańcucha znaków reprezentującego wiadomość oraz serializowanego obiektu ArrayList (zawierającego zapamiętane stany wszystkich pól wyboru).

```
import java.awt.*;
import javax.swing.*;
import java.io.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.event.*;

public class MuzMachineKoncowa implements MetaEventListener {

    JFrame ramkaGlowna;
    JPanel panelGlowny;
    JList listaOtrzymanych;
    JTextField komunikatUzytkownika;
    ArrayList<JCheckBox> listaPolWyboru;
    int nastepnyNum;
    Vector<String> wektorLista = new Vector<String>();
    String uzytkownik;
    ObjectOutputStream wyj;
    ObjectInputStream wej;
    HashMap<String, boolean[]> mapaOdebranychKompozycji = new HashMap<String, boolean[]>();

    Sequencer sekwenser;
    Sequence sekwencja;
    Sequence mojaSekwencja = null;
    Track sciezka;

    String[] nazwyInstrumentow = {"Bass Drum", "Closed Hi-Hat", "Open Hi-Hat", "Acoustic Snare", "Crash Cymbal", "Hand Clap", "High Tom", "Hi Bongo", "Maracas", "Whistle", "Low Conga", "Cowbell", "Vibraslap", "Low-mid Tom", "High Agogo", "Open Hi Conga"};

    int[] instrumenty = {35, 42, 46, 38, 49, 39, 50, 60, 70, 72, 64, 56, 58, 47, 67, 63};
```

```

public static void main (String[] args) {
    new MuzMachinaKoncowa().konfigurujAplik(args[0]); // args[0] zawiera Twoją nazwę lub identyfikator
}

public void konfigurujAplik(String nazwa) {
    użytkownik = nazwa;
    // nawiązujemy połaczenie z serwerem
    try {
        Socket sock = new Socket("127.0.0.1", 4242);
        wyj = new ObjectOutputStream(sock.getOutputStream());
        wej = new ObjectInputStream(sock.getInputStream());
        Thread watekZd = new Thread(new ZdalnyCzytelnik());
        watekZd.start();
    } catch(Exception ex) {
        System.out.println("brak połączenia - będziesz musiał grać sam.");
    }

    konigurujMidi();
    tworzGUI();
} // koniec konfiguracji

public void tworzGUI() {
    ramkaGlowna = new JFrame("MuzMachina");
    ramkaGlowna.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    BorderLayout uklad = new BorderLayout();
    JPanel panelTla = new JPanel(uklad);
    panelTla.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

    listaPolWyboru = new ArrayList<JCheckBox>();

    Box obszarPrzyciskow = new Box(BoxLayout.Y_AXIS);
    JButton start = new JButton("Start");
    start.addActionListener(new MojStartListener());
    obszarPrzyciskow.add(start);

    JButton stop = new JButton("Stop");
    stop.addActionListener(new MojStopListener());
    obszarPrzyciskow.add(stop);

    JButton tempoG = new JButton("Szybciej");
    tempoG.addActionListener(new MojTempoGListener());
    obszarPrzyciskow.add(tempoG);

    JButton tempoD = new JButton("Wolniej");
    tempoD.addActionListener(new MojTempoDListener());
    obszarPrzyciskow.add(tempoD);

    JButton wyslij = new JButton("Wyślij");
    wyslij.addActionListener(new MojWyslijListener());
    obszarPrzyciskow.add(wyslij);

    komunikatUzytkownika = new JTextField();
}

```

Dodaliśmy argument wywołania programu, przy użyciu którego można określić nazwę użytkownika.

Nic nowego... konfiguracja obsługi sieci, operacji wejścia-wyjścia oraz uruchomienie wątku odbierającego komunikaty z serwera.

Kod tworzący graficzny interfejs użytkownika, nic nowego...

Ostateczna wersja kodu aplikacji MuzMachine

```
obszarPrzyciskow.add(komunikatUzytkownika);

listaOtrzymany = new JList();
listaOtrzymany.addListSelectionListener(new WyborZListyListener());
listaOtrzymany.setSelectionMode(ListSelectionMode.SINGLE_SELECTION);
JScrollPane lista = new JScrollPane(listaOtrzymany);
obszarPrzyciskow.add(lista);
listaOtrzymany.setListData(wektorLista); // na początku brak danych
```

Box obszarNazw = new Box(BoxLayout.Y_AXIS);
for (int i = 0; i < 16; i++) {
 obszarNazw.add(new Label(nazwyInstrumentow[i]));
}

```
panelTla.add(BorderLayout.EAST, obszarPrzyciskow);
panelTla.add(BorderLayout.WEST, obszarNazw);

ramkaGlowna.getContentPane().add(panelTla);
GridLayout siatkaPolWyboru = new GridLayout(16,16);
siatkaPolWyboru.setVgap(1);
siatkaPolWyboru.setHgap(2);
panelGlowny = new JPanel(siatkaPolWyboru);
panelTla.add(BorderLayout.CENTER, panelGlowny);

for (int i = 0; i < 256; i++) {
    JCheckBox c = new JCheckBox();
    c.setSelected(false);
    listaPolWyboru.add(c);
    panelGlowny.add(c);
} // koniec pętli

ramkaGlowna.setBounds(50,50,300,300);
ramkaGlowna.pack();
ramkaGlowna.setVisible(true);
} // koniec metody
```

```
public void konigurujMidi() {
    try {
        sekvenser = MidiSystem.getSequencer();
        sekvenser.open();
        sekvenser.addMetaEventListerner(this);
        sekwencja = new Sequence(Sequence.PPQ,4);
        sciezka = sekwencja.createTrack();
        sekvenser.setTempoInBPM(120);
    } catch(Exception e) {e.printStackTrace();}
} // koniec metody
```



JList to komponent, którego do tej pory nie używaliśmy. To właśnie w nim są wyświetlane odbierane wiadomości. Jednak w odróżnieniu od normalnych pogawędek, w których można jedynie OGLĄDAĆ wiadomości, w naszej aplikacji można WYBRAĆ wiadomość z listy, aby wczytać dołączoną do niej kompozycję i ją odtworzyć.

Na tej stronie nie ma już żadnych zmian w porównaniu z poprzednimi wersjami programu.

Pobieramy sekvenser, tworzymy sekwencję i ścieżkę.

```

public void utworzSciezkeI0dtworz() {
    ArrayList<Integer> listaSciezki = null; // ta tablica będzie zawierać instrumenty
    sekwencja.deleteTrack(sciezka);
    sciezka = sekwencja.createTrack();

    for (int i = 0; i < 16; i++) {

        listaSciezki = new ArrayList<Integer>();

        for (int j = 0; j < 16; j++) {
            JCheckBox jc = (JCheckBox) listaPolWyboru.get(j + (16*i));
            if (jc.isSelected()) {
                int key = instrumenty[i];
                listaSciezki.add(new Integer(key));
            } else {
                listaSciezki.add(null); // gdyż ten wpis ścieżki powinien być pusty
            }
        } // koniec wewnętrznej pętli for
        utworzSciezke(listaSciezki);
    } // koniec zewnętrznej pętli for
    sciezka.add(tworzZdarzenie(192,9,1,0,15)); // - zawsze mamy pełne 16 taktów
    try {
        sekvenser.setSequence(sekwencja);
        sekvenser.setLoopCount(sekvenser.LOOP_CONTINUOUSLY);
        sekvenser.start();
        sekvenser.setTempoInBPM(120);
    } catch (Exception e) {e.printStackTrace();}
} // koniec metody

public class MojStartListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        utworzSciezkeI0dtworz();
    } // koniec metody actionPerformed
} // koniec klasy wewnętrznej

public class MojStopListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        sekvenser.stop();
    } // koniec metody actionPerformed
} // koniec klasy wewnętrznej

public class MojTempoGListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        float wspTempa = sekvenser.getTempoFactor();
        sekvenser.setTempoFactor((float)(wspTempa * 1.03));
    } // koniec metody actionPerformed
} // koniec klasy wewnętrznej

```

Tworzymy ścieżkę, przeglądając wszystkie pola wyboru, określając ich stan i kollarząc je z odpowiednimi instrumentami (jednocześnie tworzymy zdarzenia MidiEvent). Operacje te są dość złożone, lecz jednocześnie są IDENTYCZNE z analogicznymi czynnościami wykonywanymi we wcześniejszych wersjach programu; zatem, jeśli szukasz pełnego opisu tych operacji, zajrzyj do wcześniejszych „Kodów od kuchni”.

Odbiorcy zdarzeń GUI.
Identycznie jak w poprzednich wersjach programu.

Ostateczna wersja kodu aplikacji MuzMachine

```
public class MojTempoDListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        float wspTempa = sekwenser.getTempoFactor();
        sekwenser.setTempoFactor((float)(wspTempa * .97));
    }
}

public class MojWyslijListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        // tworzymy tablicę ze stanami pól wyboru
        boolean[] stanPolaWyboru = new boolean[256];
        for (int i = 0; i < 256; i++) {
            JCheckBox pole = (JCheckBox) listaPolWyboru.get(i);
            if (pole.isSelected()) {
                stanPolaWyboru[i] = true;
            }
        } // koniec pętli for
        String komunikatDoWyslania = null;
        try {
            wyj.writeObject(uzytkownik + nastepnyNum++ + ":" + komunikatUzytkownika.getText());
            wyj.writeObject(stanPolaWyboru);
        } catch(Exception ex) {
            System.out.println("Przykro mi bracie. Nie mogłem wysłać kompozycji na serwer.");
        }
        komunikatUzytkownika.setText("");
    } // koniec metody
} // koniec klasy wewnętrznej

To jest coś nowego... ten fragment przypomina kod programu ProstyKlientPogawedek, z tą różnicą, że zamiast wysyłania wiadomości tekstowej serializujemy dwa obiekty (String reprezentujący wiadomość oraz kompozycję) i zapisujemy je w wyjściowym strumieniu gniazda (przesyłając je tym samym na serwer).
```

```
public class WyborZListyListener implements ListSelectionListener {
    public void valueChanged(ListSelectionEvent le) {
        if (!le.getValueIsAdjusting()) {
            String wybranaOpcja = (String) listaOtrzymanych.getSelectedValue();
            if (wybranaOpcja != null) {
                // a teraz, wracamy do mapy i zmieniamy sekwencję
                boolean[] stanZaznaczonego = (boolean[]) mapaOdebranychKompozycji.get(wybranaOpcja);
                zmienSekwencje(stanZaznaczonego);
                sekwenser.stop();
                utworzSciezkeIOdtworz();
            }
        }
    } // koniec metody
} // koniec klasy wewnętrznej
```

Także ten fragment kodu jest nowy. Odbiorca zdarzeń WyborZListyListener informuje nas, kiedy użytkownik zaznaczył na liście jedną z odebranych wiadomości. Gdy to nastąpi, BEZWŁOCZNIE ładujemy kompozycję skojarzoną z wybraną wiadomością (znajduje się ona w kolekcji HashMap, w zmiennej o nazwie mapaOdebranychKompozycji) i odtwarzamy ją. W kodzie musiły znaleźć się dodatkowe instrukcje warunkowe, związane ze szczególnym sposobem obsługi zdarzeń ListSelectionEvent.

```

public class ZdalnyCzytelnik implements Runnable {
    boolean[] stanPolaWyboru = null;
    String prezentowanaNazwa = null;
    Object obj = null;
    public void run() {
        try {
            while((obj=wej.readObject()) != null) {
                System.out.println("pobraliśmy obiekt z serwera");
                System.out.println(obj.getClass());
                String nazwaDoWyswietlenia = (String) obj;
                stanPolaWyboru = (boolean[]) wej.readObject();
                mapaOdebranychKompozycji.put(nazwaDoWyswietlenia, stanPolaWyboru);
                wektorLista.add(nazwaDoWyswietlenia);
                listaOtrzymanych.setListData(wektorLista);
            } // koniec while
        } catch(Exception ex) {ex.printStackTrace();}
    } // koniec metody run
} // koniec klasy wewnętrznej

public class OdtworzMojelistener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        if (mojaSekwencja != null) {
            sekwencja = mojaSekwencja;      // przywracamy mój oryginał
        }
    } // koniec metody
} // koniec klasy wewnętrznej

public void zmienSekwencje(boolean[] stanPolaWyboru) {
    for (int i = 0; i < 256; i++) {
        JCheckBox pole = (JCheckBox) listaPolWyboru.get(i);
        if (stanPolaWyboru[i]) {
            pole.setSelected(true);
        } else {
            pole.setSelected(false);
        }
    } // koniec pętli for
} // koniec metody

public void utworzSciezke(ArrayList list) {
    Iterator iter = list.iterator();
    for (int i = 0; i < 16; i++) {
        Integer num = (Integer) iter.next();
        if (num != null) {
            int numKlaw = num.intValue();
            sciezka.add(tworzZdarzenie(144,9,numKlaw, 100, i));
            sciezka.add(tworzZdarzenie(128,9,numKlaw,100, i + 1));
        }
    } // koniec pętli for
} // koniec metody

```

Oto zadanie realizowane przez wątek — ma odczytywać dane z serwera. W tym przypadku „danymi” zawsze będą dwa serializowane obiekty: obiekt String zawierający wiadomość oraz kompozycja (czyli obiekt ArrayList przechowujący stany pól wyboru).

Po odebraniu wiadomości odczytujemy (deserializujemy) oba obiekty (String oraz tablicę boolean[]) reprezentującą stany pól wyboru) i dodajemy je do komponentu JList. Wyświetlenie czegoś w komponentie JList jest operacją dwuetapową — w pierwszym kroku należy dodać to, co chcemy wyświetlić, do przechowywanej niezależnie kolekcji Vector (to starsza wersja ArrayList), natomiast w drugim kroku trzeba poinformować komponent JList, aby użył wskazanej kolekcji Vector jako źródła danych, które ma wyświetlić.

Ta metoda jest wywoływana, gdy użytkownik zaznaczy któryś z wiadomości wyświetlonych na liście. W odpowiedzi NATYCHMIAST wyświetlimy kompozycję skojarzoną z tą wiadomością.

Wszystkie czynności związane z obsługą MIDI są takie same jak we wcześniejszych wersjach programu.

Ostateczna wersja kodu aplikacji MuzMachine

```
public MidiEvent tworzZdarzenie(int plc, int kanal, int jeden, int dwa, int takt) {  
    MidiEvent zdarzenie = null;  
    try {  
        ShortMessage a = new ShortMessage();  
        a.setMessage(plc, kanal, jeden, dwa);  
        zdarzenie = new MidiEvent(a, takt);  
    } catch(Exception e) {}  
    return zdarzenie;  
} // koniec metody  
  
} // koniec klasy
```

Nic nowego. Wszystko to samo, co we wcześniejszych wersjach programu.



Zaostrz ołówek

W jaki sposób mógłbyś poprawić ten program?

Oto kilka pomysłów, które mogą Ci pomóc:

- 1) Kiedy wybierzesz kompozycję, to jakakolwiek kompozycja, nad którą aktualnie pracowałaś, zostaje bezpowrotnie usunięta. Jeśli to była nowa kompozycja (zmodyfikowana wersja jakiejś istniejącej kompozycji), masz pecha. A zatem w takiej sytuacji możesz wyświetlać okno dialogowe i pytać użytkownika, czy chce zapisać aktualną kompozycję.
- 2) Jeśli zapomnisz podać argumentu w wierszu wywołania programu, to próba jego uruchomienia skończy się zgłoszeniem wyjątku. Umieść w metodzie main() jakiś kod, który będzie sprawdzać, czy argument został podany czy nie. Jeśli użytkownik nie podał argumentu, określ jego domyślną wartość lub wyświetl komunikat informujący, że należy ponownie uruchomić program, tym razem podając nazwę użytkownika.
- 3) Zapewne fajna byłaby możliwość pozwalająca na wygenerowanie losowej kompozycji po kliknięciu jakiegoś przycisku. Potem mógłbyś zapisać kompozycję, która Ci się spodoba. Albo jeszcze lepiej — mógłbyś stworzyć opcję pozwalającą na wczytywanie kompozycji „bazowych”, które następnie można by modyfikować.

Moglibyś przygotować takie kompozycje bazowe dla utworów jazzowych, rockowych, muzyki folk i tak dalej.

Ostateczna wersja serwera aplikacji MuzMachine

Poniższy kod jest w przeważającej części identyczny z kodem serwera BardzoProstySerwerPogawedek przedstawionego w rozdziale poświęconym wątkom i obsłudze sieci. Jedyna różnica polega na tym, iż ten serwer odbiera i rozsyła dwa serializowane obiekty, a nie jeden obiekt String (choć tak się składa, że jeden z tych obiektów też jest łańcuchem znaków).

```
import java.io.*;
import java.net.*;
import java.util.*;

public class SerwerMuzyczny {

    ArrayList<ObjectOutputStream> strumienieWyjDoKlientow;

    public static void main (String[] args) {
        new SerwerMuzyczny().doRoboty();
    }

    public class ObslugaKlientow implements Runnable {

        ObjectInputStream wej;
        Socket gniazdoKlienta;

        public ObslugaKlientow(Socket socket) {
            try {
                gniazdoKlienta = socket;
                wej = new ObjectInputStream(gniazdoKlienta.getInputStream());
            } catch(Exception ex) {ex.printStackTrace();}
        } // koniec konstruktora

        public void run() {
            Object o2 = null;
            Object o1 = null;
            try {

                while ((o1 = wej.readObject()) != null) {

                    o2 = wej.readObject();

                    System.out.println("odczytano dwa obiekty");
                    przekazDoWszystkich(o1, o2);
                } // koniec while
            } catch(Exception ex) {ex.printStackTrace();}
        } // koniec run
    } // koniec klasy wewnętrznej
}
```

Ostateczna wersja kodu aplikacji MuzMachine

```
public void doRoboty() {
    strumienieWyjDoKlientow = new ArrayList<ObjectOutputStream>();

    try {
        ServerSocket gniazdoSerwera = new ServerSocket(4242);

        while(true) {
            Socket clientSocket = gniazdoSerwera.accept();
            ObjectOutputStream wyj = new ObjectOutputStream(clientSocket.getOutputStream());
            strumienieWyjDoKlientow.add(wyj);

            Thread watek = new Thread(new ObslugaKlientow(clientSocket));
            watek.start();

            System.out.println("mamy połaczenie");
        }
    } catch(Exception ex) {
        ex.printStackTrace ();
    }
} // koniec metody

public void przekazDoWszystkich(Object obj1, Object obj2) {
    Iterator iter = strumienieWyjDoKlientow.iterator();
    while(iter.hasNext()) {
        try {
            ObjectOutputStream wyj = (ObjectOutputStream) iter.next();
            wyj.writeObject(obj1);
            wyj.writeObject(obj2);
        } catch(Exception ex) {ex.printStackTrace();}
    }
} // koniec metody

} // koniec klasy
```

Dodatek B

Dziesięć najważniejszych zagadnień, które niemal znalazły się w tej książce...



W książce przedstawiliśmy wiele zagadnień i już niemal zbliżamy się do jej końca. Będziemy za Tobą tęsknić, jednak zanim się pożegnamy i wpuścimy Cię do królestwa Javy, chcielibyśmy przekazać Ci kilka dodatkowych informacji, gdyż bez tego nie moglibyśmy mieć poczucia dobrze spełnionego zadania. Zapewne nie będziemy w stanie umieścić w tym stosunkowo niewielkim dodatku wszystkiego, co mogłoby Ci się przydać. W rzeczywistości początkowo umieściliśmy w nim wszystkie informacje o Javie, jakich mógłbyś potrzebować (oczywiście te, które nie zostały opisane we wcześniejszych rozdziałach), lecz trzeba by je wydrukować czcionką o wielkości 0,000003. Wszystko by się zmieściło, lecz nikt nie byłby w stanie tego przeczytać. Dlatego też wyrzuciliśmy większość tych informacji i zostawili jedynie dziesięć najważniejszych zagadnień.

To już naprawdę koniec tej książki. Oczywiście za wyjątkiem indeksu (który koniecznie musisz przeczytać!).

Nr 10. Operacje bitowe

Dlaczego to ma mnie obchodzić?

Wspominaliśmy już, że w bajcie jest 8 bitów, w wartości typu short — 16 i tak dalej. Od czasu do czasu sytuacja może wymagać „włączania” i „wyłączania” konkretnych bitów. Na przykład możesz tworzyć oprogramowanie dla nowego tostera i dojść do wniosku, że ze względu na ograniczenia pamięci pewne jego ustawienia muszą być kontrolowane na poziomie bitów. Aby ułatwić Ci lekturę, w komentarzach nie będziemy przedstawiać pełnych 32 bitów liczby całkowitej, a jedynie 8 ostatnich.

Bitowy operator negacji (NOT): ~

Ten operator zmienia wszystkie bity wartości na „przeciwne”.

```
byte x = 10; // w postaci bitów: 00001010
x = ~x; // a teraz: 11110101
```

Kolejne trzy operatory porównują dwie wartości bit po bitie i zwracają wynik tego porównania. W przykładach będziemy używać następujących dwóch zmiennych:

```
byte x = 10; // w postaci bitów: 00001010
byte y = 6; // w postaci bitów: 00000110
```

Bitowy operator „i” (AND): &

Zwraca liczbę, której bity mają wartość 1 tylko, jeśli *oba* oryginalne bity mają wartości 1.

```
byte z = x & y; // wyniki bitowo: 00000010
```

Bitowy operator „lub” (OR): |

Zwraca liczbę, której bity mają wartość 1, jeśli *którykolwiek* z oryginalnych bitów ma wartość 1.

```
byte z = x | y; // wynik bitowo: 00001110
```

Bitowy operator alternatywy wykluczającej (XOR): ^

Zwraca liczbę, której bity mają wartość 1 tylko w przypadku, gdy *dokładnie jeden* z oryginalnych bitów ma wartość 1.

```
byte z = x ^ y; // wyniki bitowo: 00001100
```

Operatory przesunięcia

Operatory te wymagają podania jednej wartości całkowitej i przesuwają wszystkie jej bity w określonym kierunku. Jeśli zechcesz odświeżyć swoje wiadomości z zakresu arytmetyki binarnej, zauważysz na pewno, że przesuwanie wartości *w lewo* oznacza *mnożenie* jej przez potęgę liczby 2, natomiast przesuwanie *w prawo* oznacza *dzielenie* przez potęgę liczby 2. Prezentując następne trzy operatory, posłużymy się następującym przykładem:

```
byte x = -11; // w postaci bitowej: 11110101
```

No dobrze, dobrze, odkładaliśmy to do tej pory, ale oto jest — najkrótsze wyjaśnienie sposobu zapisu liczb ujemnych i dopełnienia dwójkowego. Pamiętaj, że skrajny lewy bit liczby całkowitej, jest nazywany **bitem znaku**. W przypadku liczb ujemnych w Javie bit znaku *zawsze* ma wartość 1. Z kolei bit znaku w liczbach dodatnich zawsze ma wartość 0. W celu przechowywania wartości ujemnych w Javie wykorzystywany jest wzór *dopełnienia dwójkowego*. Aby zmienić znak wartości na przeciwny, należy zmienić wszystkie jej bity na „przeciwne”, a następnie do uzyskanego wyniku dodać 1 (w przypadku wartości typu byte oznaczałoby to dodanie do wyniku uzyskanego w wyniku zamiany bitów wartości 00000001).

Operator przesunięcia w prawo: >>

Operator ten przesuwa wszystkie bity liczby o podaną ilość „miejsc”. Skrajne lewe bity są uzupełniane tą samą wartością, jaką początkowo miał skrajny lewy bit.

Zatem wartość bitu znaku *nie zmienia się*:

```
int y = x >> 2; // wynik bitowo: 11111101
```

Operator przesunięcia w prawo bez znaku: >>>

Działa podobnie jak operator przesunięcia w prawo z tą różnicą, że w skrajnych lewych bitach ZAWSZE jest zapisywane 0. A zatem bit znaku *może się zmienić*.

```
int y = x >>> 2; // wynik bitowo: 00111101
```

Operator przesunięcia w lewo: <<

Działa podobnie jak operator przesunięcia w prawo bez znaku z tą różnicą, że bity przesuwane są w kierunku przeciwnym; w skrajnych prawych bitach zawsze zapisywane są 0. Bit znaku *może się zmienić*.

```
int y = x << 2; // wynik bitowo: 11010100
```

Nr 9. Niezmienność

Dlaczego miałyby mnie obchodzić, że łańcuchy znaków są niezmienne?

Wraz z powiększaniem się programów nieuchronnie będzie się w nich pojawiać coraz więcej obiektów String. Ze względów bezpieczeństwa oraz w celu zaoszczędzenia pamięci (pamiętaj, że programy pisane w Javie muszą też działać w małych telefonach dysponujących odpowiednimi możliwościami) łańcuchy znaków w Javie są niezmienne. A to oznacza, że w przypadku użycia następującego fragmentu kodu:

```
String s = "0";
for (int x = 1; x < 10; x++) {
    s = s + x;
}
```

tak naprawdę wykonanie tego kodu będzie się wiązało z utworzeniem dziesięciu obiektów String (zawierających kolejno następujące łańcuchy znaków: "0", "01", "012" i tak dalej aż do "0123456789"). Ostatecznie zmienna s zawiera odwołanie do łańcucha "0123456789", lecz jednocześnie istnieje aż dziesięć obiektów String!

Za każdym razem, gdy tworzysz nowy obiekt String, wirtualna maszyna Javy umieszcza go w specjalnym fragmencie pamięci, nazywanym „obszarem łańcuchów znaków”. Jeśli w obszarze tym znajduje się już obiekt o takiej samej wartości, to JVM nie tworzy nowego obiektu, lecz zapisuje w zmiennej odwołanie do obiektu istniejącego. Rozwiązywanie takie jest możliwe, gdyż łańcuchy znaków są niezmienne, a zatem jedna zmienna referencyjna nie może zmienić wartości łańcucha, do którego jednocześnie odwołuje się inna zmienna.

I jeszcze jedna sprawa związana z obszarem łańcuchów znaków — otóż odśmieczac *nie zajmuje się tym obszarem pamięci*. W naszym przypadku oznaczałoby to, że jeśli w dalszej części programu nie pojawi się np. łańcuch o wartości "01234", to pierwsze dziewięć łańcuchów utworzonych w pętli for będzie niepotrzebnie zajmować pamięć.

W jaki sposób to rozwiązanie może pomagać w oszczędzaniu pamięci?

Cóż, jeśli nie będziesz uważny, to w ogóle nie pomoże! Jeśli jednak dobrze zrozumiesz zasadę niezmienności łańcuchów znaków, to czasami będziesz mógł ją wykorzystać, aby

zaoszczędzić nieco pamięci. Jeśli jednak musisz wykonywać wiele operacji na łańcuchach znaków (na przykład łączyć łańcuchy), to istnieje inna klasa, która doskonale nadaje się do tego celu — jest to klasa StringBuffer. Opiszymy ją dokładniej w dalszej części dodatku.

Dlaczego miałyby nas obchodzić, że klasy reprezentujące typy podstawowe są niezmienne?

Podstawowe dwa zastosowania klas reprezentujących podstawowe typy danych (tak zwanych klas „opakowujących”) zostały podane w rozdziale poświęconym operacjom matematycznym i klasie Math.

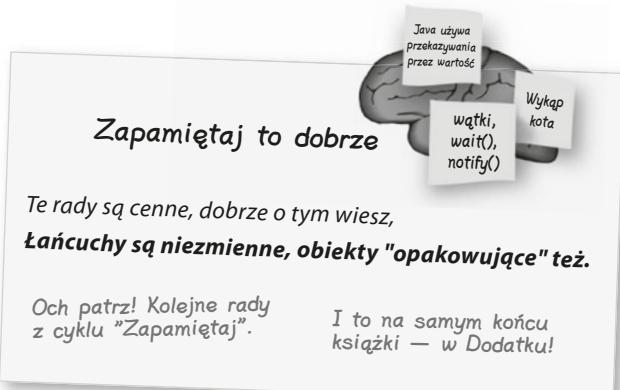
Są to:

- Zapewnienie możliwości, by wartość typu podstawowego udawała obiekt.
- Zapewnienie możliwości użycia statycznych metod pomocniczych (takich jak Integer.parseInt()).

Koniecznie należy zapamiętać, że tworząc obiekt takiej klasy, na przykład przy użyciu poniższej instrukcji:

```
Integer iObj = new Integer(42);
```

uzyskamy obiekt, którego wartości nie będziemy już mogli zmienić — zawsze będzie nią 42. **Klasy „opakowujące” nie udostępniają żadnych metod ustawiających i zwracających**. Oczywiście będziemy mogli się odwołać do obiektu iObj i przypisać mu wartość innego obiektu, ale w takim przypadku uzyskamy *dwa* obiekty. Kiedy utworzymy obiekt reprezentujący wartość typu podstawowego, już nigdy nie będziemy mogli zmienić jego wartości!



Asercje

Nr 8. Asercje

W niniejszej książce nie poświęciliśmy zbyt wiele uwagi zagadnieniu testowania programów pisanych w Javie na etapie ich tworzenia. Uważamy, że powinieneś uczyć się Javy, korzystając z możliwości, jakie dają narzędzia obsługiwane z poziomu wiersza poleceń. Kiedy już będziesz ekspertem, to jeśli zdecydujesz się na korzystanie z IDE¹, będziesz miał do dyspozycji także inne narzędzia do testowania. W dawnych czasach, gdy programista Javy chciał przetestować tworzony program, umieszczał w jego kodzie serię wywołań `System.out.println()` wyświetlających wartości zmiennych oraz komunikat „Jesteśmy tutaj”, dzięki którym mógł sprawdzić, czy przebieg realizacji programu jest poprawny. (Przykłady takiego zastosowania metody `println()` można znaleźć w „Kodzie gotowym do użycia” w rozdziale 6.). Kiedy program działa już jak należy, programista ponownie przeglądał cały kod programu i usuwał wszystkie „testowe” wywołania `System.out.println()`. Rozwiązań to było uciążliwe i mogło się przyczyniać do powstawania błędów. Jednak w Javie 1.4 testowanie stało się znacznie prostsze. Dzięki czemu?

Asercje

Asercje to jak gdyby metody `System.out.println()` działające pod wpływem środków dopingujących. Można dodawać je do kodu w taki sam sposób jak wywołania metody `println()`. Kompilator zakłada, że kompilowane kody źródłowe są zgodne z wersją 5.0 Javy, dlatego też w tej wersji języka mechanizmy obsługi asercji są domyślnie włączone.

Jeśli nie zaznaczyłeś chęci wykorzystania asercji, to podczas wykonywania programu instrukcje `assert` umieszczone w kodzie będą ignorowane i nie spowodują spowolnienia jego pracy. Jeśli jednak nakażesz wirtualnej maszynie Javy włączenie mechanizmów obsługi asercji, to pomogą Ci one w testowaniu programu i to bez konieczności modyfikowania choćby jednego wiersza kodu!

Niektórzy programiści narzekają na konieczność pozostawiania kodu asercji w produkcyjnym kodzie aplikacji, jednak rozwiązanie takie jest korzystne w początkowym okresie korzystania z gotowego produktu. Jeśli Twój klient ma jakieś problemy z programem, to wystarczy poinstruować go, co należy zrobić, aby uruchomić program z włączonymi mechanizmami obsługi asercji i poprosić o przesłanie uzyskanych wyników. Gdyby instrukcje asercji zostały usunięte z produkcyjnego kodu aplikacji, to nigdy nie mógłbyś skorzystać z takiej możliwości.

Stosowanie asercji niemal nie ma wad, gdyż, jeśli nie zaznaczyłeś, że należy ich używać, wirtualna maszyna Javy całkowicie je ignoriuje, dzięki czemu nie trzeba się obawiać spadku efektywności działania programu.

Jak można zastosować asercje?

Asercje można dodawać do kodu w sytuacjach, gdy uważasz, że pewien warunek *musi być spełniony*. Na przykład:

```
assert (wysokosc > 0);
// jeśli to prawda, to program będzie dalej
// wykonywany w normalny sposób;
// jeśli warunek nie zostanie spełniony,
// zostanie zgłoszony błąd AssertionError
```

Istnieje także możliwość przekazania pewnych dodatkowych danych, które zostaną dołączone do standardowych informacji o błędzie:

```
assert (wysokosc > 0) : "wysokosc = " + wysokosc +
" waga = " + waga;
```

Wyrażeniem podawanym pod dwukropku może być dowolne, poprawne wyrażenie Javy, *które zwraca wynik różny od null*. Jednak, niezależnie od tego co robisz, *nigdy nie powinieneś tworzyć asercji, które zmieniają stan obiektów!* W takim przypadku włączenie asercji podczas wykonywania programu może zmienić sposób jego działania.

Kompilacja i wykonywanie programu z wykorzystaniem asercji

Oto jak należy skompilować program, by były wykorzystywane asercje:

```
javac GraTestowa.java
```

(Zwróć uwagę, że nie trzeba podawać żadnych dodatkowych parametrów).

Poniżej pokazany został sposób *uruchamiania* programu z wykorzystaniem mechanizmów obsługi asercji:

```
java -ea GraTestowa.java
```

¹ IDE to skrót od słów Integrated Development Environment oznaczających zintegrowane środowisko programistyczne. Do narzędzi tego typu można zaliczyć Eclipse (eclipse.org), JBuilder firmy Borland oraz NetBeans (netbeans.org).

Nr 7. Zasięg blokowy

W rozdziale 9. wspominaliśmy, że zmienne lokalne istnieją jedynie tak długo, jak długo metoda, w której zostały zadeklarowane, jest umieszczona na stosie. Jednak czas „życia” niektórych zmiennych może być jeszcze krótszy. Otóż wewnątrz metod czasami tworzone są *bloki kodu*. Robiliśmy tak niemal od samego początku książki, choć nigdy jawnie nie wspominaliśmy o blokach. Zazwyczaj bloki kodu występują wewnątrz metod, a ich granice wyznaczają pary nawiasów klamrowych — { }. Często spotykamy przykładami bloków kodu są pętle (for oraz while) czy instrukcje warunkowe (if).

Spójrzmy na poniższy przykład:

```
void zrobCos() { ← Początek bloku metody.
    int x = 0; ← Zmienna lokalna dostępna w całej metodzie.
    for (int y = 0; y < 5; y++) { ← Początek bloku pętli for,
        x = x + y; ← Nie ma problemu, w tym miejscu kodu
    } ← Koniec bloku pętli.                               dostępna jest zarówno zmienna x, jak i y.

    x = x * y; ← Ups! Tego nie da się skompilować! Znajdujemy się
} ← Koniec bloku metody. W tym momencie          poza zasięgiem, w którym jest dostępna zmienna y!
                                                (Pamiętaj jednak, że ta zasada nie obowiązuje we
                                                wszystkich językach programowania!)          dostępna zmienna x.
```

W powyższym przykładzie zmienna y była zmienną „blokową”, została zadeklarowana wewnątrz bloku i stawała się niedostępna, gdy tylko zakończona została pętla for. Twoje programy pisane w Javie mogą ułatwiać testowanie i późniejszą rozbudowę, jeśli, gdzie to tylko możliwe, będziesz się starał stosować zmienne lokalne zamiast składowych i zmienne „blokowe” zamiast zmiennych lokalnych. Kompilator zadba o to, abyś nie mógł wykorzystać zmiennej, która nie jest dostępna w danym zasięgu, zatem nie musisz się obawiać występowania błędów podczas działania programu.

Nr 6. Połączone wywołania

Choć w niniejszej książce omawialiśmy złożone zagadnienia, to jednak staraliśmy się, aby kod prezentowanych programów był jak najbardziej przejrzysty i łatwy do zrozumienia. Niemniej jednak istnieją w Javie pewne możliwości skracania kodu, z którymi na pewno się spotkasz, zwłaszcza jeśli będziesz musiał analizować dużo kodu, którego sam nie napisał. Jednym z najczęściej spotykanych sposobów skracania kodu są tak zwane *połączone wywołania*. Na przykład:

```
StringBuffer sb = new StringBuffer("własna");
sb = sb.delete(3,6).insert(2,"ichur").deleteCharAt(1);
System.out.println("") + sb);
// wynik: wichura
```

Ale co się dzieje w drugim wierszu tego kodu? Oczywiście jest to dosyć nierealistyczny przykład, lecz musisz się nauczyć, jak go należy interpretować.

1. Zaczynaj od lewej.
2. Określ wynik skrajnego lewego wywołania metody, w tym przypadku, jest to wynik wywołania `sb.delete(3,6)`. Jeśli zajrzesz do dokumentacji API, przekonasz się, że metoda `delete()` zwraca obiekt `StringBuffer`. Zatem w wyniku wywołania tej metody uzyskujemy obiekt `StringBuffer` zawierający znaki "ała".
3. Wykonywane jest drugie od lewej wywołanie (w tym przypadku jest to metoda `insert()`), przy czym jest w nim używany obiekt `StringBuffer` zwrócony przez pierwsze wywołanie. *Także* i to wywołanie (metody `insert()`) zwraca obiekt `StringBuffer` (choć typ zwracanego obiektu nie musi być taki sam jak w pierwszym wywołaniu). I właśnie w ten sposób jest wykonywane to wyrażenie — obiekt zwrócony przez pierwsze wywołanie, jest używany do wykonania kolejnej metody (zaczynając od lewej i przesuwając się ku prawej stronie wyrażenia). Teoretycznie rzecz biorąc, w jednej instrukcji można połączyć dowolnie dużo wywołań (choć w praktyce rzadko kiedy spotyka się ich więcej niż trzy).

Bez możliwości łączenia wywołań drugi wiersz kodu trzeba by zapisać w następujący sposób:

```
sb = sb.delete(3,6);
sb = sb.insert(2,"ichur");
sb = sb.deleteCharAt(1);
```

Jednak można podać częściej spotykany przykład łączenia wywołań — używaliśmy go w tej książce i sądziliśmy, że nam o nim przypomnisz. Stosuje się go w sytuacji, gdy metoda `main()` musi wywołać metodę głównej klasy programu, jednak przechowywanie *odwołania* do obiektu tej klasy nie jest konieczne. Innymi słowy, obiekt klasy głównej jest konieczny *wyłącznie* po to, aby można było wywołać jedną z jego metod.

```
class Tmp {
    public static void main(String[] args) {
        new Tmp().doRoboty();      ←
    }
}

void doRoboty() {
    // tu wykonujemy to, na czym nam naprawdę zależy
}
```

Chcemy wywołać metodę `doRoboty()`, ale nie interesuje nas obiekt `Tmp`, zatem nie zaprzatamy sobie głowy tym, aby zapisywać go w jakiejś zmiennej.

Nr 5. Klasy zagnieżdżone — anonimowe i statyczne

Istnieje wiele rodzajów klas zagnieżdżonych

W części książki poświęconej obsłudze graficznego interfejsu użytkownika zaczęliśmy używać klas wewnętrznych (zagnieżdżonych), które umożliwiały nam implementowanie interfejsów obsługi zdarzeń. Jest to najczęściej spotykany, praktyczny i najbardziej czytelny sposób wykorzystania klas wewnętrznych — klasa jest w całości umieszczona wewnątrz nawiasów klamrowych wyznaczających inną klasę. Pamiętaj jednak, że w takim przypadku klasa wewnętrzna jest *składową* klasy zewnętrznej, dlatego też, aby się do niej dostać, musimy dysponować odwołaniem do obiektu klasy zewnętrznej.

Istnieją jednak także inne rodzaje klas wewnętrznych — *statyczne* oraz *anonimowe*. Nie będziemy się tu wdawać w niepotrzebne szczegóły, lecz nie chcemy także, byś był zaskoczony nieznanymi konstrukcjami składniowymi, gdybyś je zobaczył w kodzie napisanym przez kogoś innego. Ponieważ spośród wszystkich konstrukcji składniowych języka Java chyba żadne nie pozwalają tworzyć tak dziwnie wyglądającego kodu, jak właśnie anonimowe klasy wewnętrzne. Zaczniemy jednak od czegoś prostszego — statycznych klas zagnieżdżonych.

Statyczne klasy zagnieżdżone

Już wiesz, co oznacza termin „statyczny” — coś jest związane z klasą, a nie z jej konkretnym obiektem. Zagnieżdżone klasy statyczne wyglądają podobnie do „niestatycznych” klas zagnieżdżonych, których używaliśmy do obsługi zdarzeń, przy czym są dodatkowo oznaczone słowem kluczowym `static`.

```
public class PawZewnetrzny {
    static class LewLewnetrzny {
        void wypowiedz() {
            System.out.println("Metoda statycznej klasy wewnętrznej.");
        }
    }
}

class Test {
    public static void main(String[] args) {
        PawZewnetrzny.LewLewnetrzny tmp = new PawZewnetrzny.LewLewnetrzny();
        tmp.wypowiedz();
    }
}
```

Statyczna klasa zagnieżdżona jest dokładnie tym, co sugeruje nazwa — klasa, której definicja jest umieszczona wewnątrz innej klasy i poprzedzona słowem kluczowym `static`.

Ponieważ statyczna klasa zagnieżdżona jest... statyczna, dlatego nie używamy obiektu klasy zewnętrznej. Wystarczy podać nazwę klasy zewnętrznej dokładnie w taki sam sposób, w jaki wywołujemy metody statyczne lub odwołujemy się do statycznych składowych.

Statyczne klasy zagnieżdżone przypominają zwyczajne — „niezagnieżdżone” — klasy pod tym względem, że nie dysponują szczególnym związkiem z obiektem klasy zewnętrznej. Jednak wciąż są one uważane za *składowe* klasy zewnętrznej, dlatego też dysponują dostępem do ich składowych prywatnych, jednak... *tylko tych, które także są statyczne*. Ponieważ statyczne klasy zagnieżdżone nie są powiązane z obiektem klasy zewnętrznej, nie mają zatem żadnej możliwości odwołania się do zwyczajnych (niestatycznych) składowych i metod tego obiektu.

Nr 5. Klasy zagnieżdżone — anonimowe i statyczne; ciąg dalszy

Różnica pomiędzy zagnieżdżoną i wewnętrzna

W Javie klasa zdefiniowana w zakresie innej klasy jest nazywana klasą **zagnieżdżoną**. Nie ma żadnego znaczenia, czy jest to klasa anonimowa, statyczna, normalna, czy jakakolwiek inna. Jeśli tylko została umieszczona wewnętrz w innej klasie, to z technicznego punktu widzenia jest uznawana za klasę zagnieżdżoną. Jednak *niestatyczne* klasy zagnieżdżone są często określane jako klasy **wewnętrzne** — właśnie tak nazywaliśmy je we wcześniejszej części tej książki. Wniosek: wszystkie klasy wewnętrzne są klasami zagnieżdżonymi, jednak nie wszystkie klasy zagnieżdżone są wewnętrzny.

Anonimowe klasy wewnętrzne

Wyobraź sobie, że piszesz kod związanego z graficznym interfejsem użytkownika i nagle zdajesz sobie sprawę, że będziesz potrzebował obiektu jakiejś klasy implementującej interfejs `ActionListener`. Jednocześnie uświadamiasz sobie, że jeszcze nigdy nie napisałś takiej *klasy*. W takim przypadku masz dwa wyjścia z sytuacji:

1. Dodać do swojego kodu klasę wewnętrzna w taki sam sposób, w jaki robiłeś to, tworząc kod obsługujący graficzny interfejs użytkownika, i przekazać obiekt tej klasy w wywołaniu metody rejestrującej (`addActionListener()`) przycisku.
2. Utworzyć *anonimową* klasę wewnętrzna oraz jej obiekt — dokładnie tam, gdzie jej potrzebujesz. **Dostępnie — w tym miejscu kodu, w jakim potrzebujesz obiektu obsługującego zdarzenia.** Właśnie tak, możesz utworzyć klasę i obiekt tej klasy w tym miejscu kodu, w którym normalnie posłużyłbyś się obiektem. Zastanów się nad tym przez chwilę — oznacza to, że w miejscu, gdzie normalnie przekazywałbyś *obiekt* jako argument wywołania metody, teraz będziesz przekazywał całą *klasę*!

```
import java.awt.event.*;
import javax.swing.*;

public class TestAnon {
    public static void main (String[] args) {

        JFrame frame = new JFrame();
        JButton button = new JButton("click");
        frame.getContentPane().add(button);
        // button.addActionListener(quitListener);

        Ta instrukcja:
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ev) {
                System.exit(0);
            }
        });
        } kończy się
        aż tutaj!
    }

    Zauważ, że użyliśmy zapisu „new ActionListener()”, chociaż ActionListener jest interfejsem, a przecież nie można TWORZYĆ obiektu interfejsu! Jednak w rzeczywistości zastosowana składnia oznacza: „utwórz nową klasę (nieposiadającą żadnej nazwy) implementującą interfejs ActionListener... a swoją drogą, tu jest implementacja metody actionPerformed() tego interfejsu”.
}
```

Utworzyliśmy ramkę i dodaliśmy do niej przycisk, teraz musimy zarejestrować odbiorcę, który będzie obsługiwać zdarzenia generowane przez ten przycisk. Z tym że nie stworzyliśmy jeszcze żadnej klasy implementującej interfejs `ActionListener`...

Normalnie zrobilibyśmy coś takiego — przekazalibyśmy odwołanie do obiektu klasy wewnętrznej... czyli klasy implementującej interfejs `ActionListener` (oraz jego metodę `actionPerformed()`).

Jednak tym razem, zamiast przekazywać odwołanie do obiektu, przekazujemy... definicję zupełnie nowej klasy!! Innymi słowy, piszemy klasę implementującą interfejs `ActionListener` DOKŁADNIE W TYM MIEJSCU, GDZIE JEST NAM ONA POTRZEBNA. Taki zapis powoduje także automatyczne utworzenie obiektu tej nowej klasy.

Nr 4. Poziomy dostępu oraz modyfikatory dostępu (czyli co kto widzi)

Java ma cztery poziomy dostępu oraz trzy modyfikatory dostępu. Istnieją tylko trzy modyfikatory dostępu, gdyż jeden z poziomów dostępu jest stosowany domyślnie (czyli zostanie użyty, jeśli nie podamy żadnego modyfikatora).

Poziomy dostępu (wymienione w kolejności od najmniej do najbardziej restrykcyjnych):

publiczny ← oznacza, że dowolny kod umieszczony w dowolnym miejscu programu będzie mieć dostęp do publicznego „elementu” (przy czym „elementem” może być klasa, zmienna, metoda, konstruktor itp.).

chroniony ← działa tak samo, jak poziom domyślny (czyli dostęp do elementu ma kod umieszczony w tym samym pakiecie), JEDNAK także klasy potomne umieszczone poza pakietem mogą dziedziczyć elementy chronione.

domyślny poziom dostępu ← oznacza, że do „elementu” będzie mieć dostęp wyłącznie kod należący do tego samego pakietu.

prywatny ← dostęp do prywatnego „elementu” będzie mieć wyłącznie kod należący do tej samej klasy. Pamiętaj, iż ten poziom dostępu oznacza, że „element” jest prywatny w ramach klasy, ale nie dla konkretnego obiektu. Jeden obiekt Pies będzie mieć dostęp do prywatnych „elementów” innego obiektu Pies, jednak obiekt Kot nie będzie mógł z nich korzystać.

Modyfikatory dostępu

public
protected
private

W przeważającej większości przypadków będziesz używać wyłącznie modyfikatorów **public** oraz **private**.

public

Tego modyfikatora możesz używać do określania poziomu dostępu klas, stałych (statycznych zmiennych finalnych), konstruktorów oraz metod, które chcesz udostępnić dla innego kodu (na przykład metod ustawiających i zwracających).

private

Niemal wszystkie składowe klas powinny być prywatne. Dotyczy to także tych metod, które nie powinny być wywoływanie przez kod spoza klasy (innymi słowy metod, które są *używane* przez publiczne metody klasy).

Chociaż nie musisz korzystać z pozostałych dwóch modyfikatorów dostępu, to jednak powinieneś wiedzieć, do czego służą, gdyż na pewno się z nimi niejeden raz spotkasz.

Nr 4. Poziomy dostępu i modyfikatory dostępu, ciąg dalszy

default oraz protected

default

Zarówno chroniony, jak i domyślny poziom dostępu są ściśle związane z pakietami. Poziom domyślny jest łatwy — oznacza on, że jedynie kod *należący do tego samego pakietu* może skorzystać z kodu o domyślnym poziomie dostępu. Przykładowo oznacza to, że dostęp do klasy o domyślnym poziomie dostępu (czyli klasy, której jawnie nie oznaczono jako *publicznej*) mogą uzyskać wyłącznie inne klasy należące do tego samego pakietu.

Ale co to oznacza mieć *dostęp* do klasy? Otóż kod, który nie ma dostępu do klasy, nie może nawet o niej *pomyśleć*. A w tym przypadku „myśleć” oznacza „używać kodu danej klasy”. Na przykład, jeśli nie mamy dostępu do klasy, to nie możemy tworzyć jej obiektów ani nawet deklarować zmiennych, argumentów ani wartości wynikowych tego typu. Po prostu w ogóle nie możemy wpisać nazwy tej klasy w naszym kodzie. Jeśli to zrobisz, kompilator zwróci błąd.

Przeanalizuj implikacje domyślnego poziomu dostępu — jeśli pewna klasa o dostępie domyślnym będzie mieć publiczne metody, to tak naprawdę metody te wcale nie będą publiczne. Nie możemy bowiem używać metod, jeśli nie mamy *dostępu* do samej klasy.

Ale niby dlaczego ktoś chciałby ograniczać dostęp do klasy i udzielać go tylko klasom należącym do tego samego pakietu? Zazwyczaj pakiety są projektowane jako grupy współpracujących, powiązanych ze sobą klas. A zatem rozwiązanie, w którym klasy należące do tego samego pakietu mają dostęp do swojego kodu, lecz jedynie niewielka część klas i metod całego pakietu jest dostępna dla „światu zewnętrznego” (czyli kodu spoza pakietu), jest całkiem sensowne.

No dobrze, tak wygląda dostęp domyślny. Jak już wspominaliśmy, jest on bardzo prosty — jeśli „coś” ma dostęp domyślny (a to oznacza, że nie użyto żadnego modyfikatora dostępu), to jedynie kod należący do tego samego pakietu będzie mieć dostęp do tego „czegoś” (klasy, metody, zmiennej, klasy wewnętrznej).

Zatem do czego służy dostęp *chroniony*?

protected

Dostęp chroniony jest niemal taki sam jak dostęp domyślny. Z jednym jedynym wyjątkiem — pozwala klasom potomnym na *dziedziczenie „elementów”* chronionych, *nawet jeśli klasy te należą do innego pakietu niż klasa bazowa*. I to wszystko. Tylko tyle daje nam dostęp chroniony — możliwość, by klasy potomne znajdowały się poza pakietem klasy bazowej, a pomimo to, by *dziedziczyły* różne „elementy” klasy bazowej, w tym także jej konstruktory i metody.

Rzadko kiedy programiści znajdują powody, by korzystać z chronionego poziomu dostępu, jednak w niektórych przypadkach może się okazać, że jest to właśnie to, czego Ci trzeba. Jednym z ciekawych aspektów tego poziomu dostępu, aspektem, który odróżnia go od pozostałych poziomów dostępu, jest fakt, iż ma on związek wyłącznie z *dziedziczeniem*. Jeśli klasa potomna znajdująca się poza pakietem klasy bazowej posiada *odwołanie* do obiektu klasy bazowej (która, założymy, ma metodę chronioną), to posługując się tym odwołaniem z poziomu obiektu klasy potomnej, nie będzie można wywołać chronionej metody klasy bazowej! Jedynym sposobem, w jaki klasa potomna może uzyskać dostęp do tej metody, jest jej *oddziedziczenie*. Innymi słowy, klasa potomna znajdująca się poza pakietem klasy bazowej nie ma *dostępu* do chronionej metody swojej klasy bazowej, jednak dzięki dziedziczeniu *posiada* tę metodę.

Nr 3. Metody klas **String** oraz **StringBuffer**

Dwie naj częściej stosowanymi klasami całego Java API są **String** oraz **StringBuffer** (pamiętasz zapewne z zagadnienia numer 9., opisanego we wcześniejszej części tego dodatku, że łańcuchy znaków w Javie są niezmienne, a zatem w przypadku wykonywania wielu operacji na łańcuchach wykorzystanie klas **StringBuffer** lub **StringBuilder** może być znacznie wydajniejsze niż stosowanie klasy **String**). W wersji 5.0 Javy należy używać klasy **StringBuilder** zamiast **StringBuffer**, chyba że operacje wykonywane na łańcuchach znaków będą musiały prawidłowo działać w aplikacjach wielowątkowych; jednak taki wymóg nie pojawia się często. Poniżej, dla Twojej przyjemności, przedstawiliśmy kluczowe metody obu tych klas:

Metody dostępne w obu klasach — zarówno **String, jak i **StringBuffer/StringBuilder**:**

<code>char charAt(int indeks);</code>	// jaki jest znak w określonym miejscu łańcucha
<code>int length();</code>	// jaka jest długość łańcucha
<code>String substring(int pocz, int kon);</code>	// pobiera fragment łańcucha
<code>String toString();</code>	// jaki jest łańcuch znaków reprezentujący wartość obiektu

Metody służące do łączenia łańcuchów:

<code>String concat(łańcuchZnaków);</code>	// dla klasy String
<code>String append(String);</code>	// dla klasy StringBuffer i StringBuilder

Metody klasy **String:**

<code>String replace(char stary, char nowy);</code>	// zamienia wszystkie wystąpienia znaku
<code>String substring(int pocz, int kon);</code>	// pobiera fragment łańcucha
<code>char [] toCharArray();</code>	// zamienia łańcuch na tablicę znaków
<code>String toLowerCase();</code>	// zamienia wszystkie litery łańcucha na małe
<code>String toUpperCase();</code>	// zamienia wszystkie litery łańcucha na duże
<code>String trim();</code>	// usuwa wszystkie białe znaki z obu końców łańcucha
<code>String valueOf(char []);</code>	// zamienia tablicę znaków na łańcuch
<code>String valueOf(int i);</code>	// tworzy ciąg o zawartości typu podstawowego
	// istnieją także wersje metody dla innych typów podstawowych

Metody klasy **StringBuffer oraz **StringBuilder**:**

<code>StringBxxx delete(int start, int koniec);</code>	// usuwa fragment łańcucha
<code>StringBxxx insert(int przes, dowolna wartość podstawowa lub char []);</code>	// wstawia daną
<code>StringBxxx replace(int start, int koniec, String s);</code>	// zastępuje fragment podanym łańcuchem
<code>StringBxxx reverse();</code>	// zmienia kolejność znaków
<code>void setCharAt(int indeks, char c);</code>	// zastępuje wskazany znak

Uwaga: Zapis `StringBxxx` oznacza klasę **StringBuffer** lub **StringBuilder**.

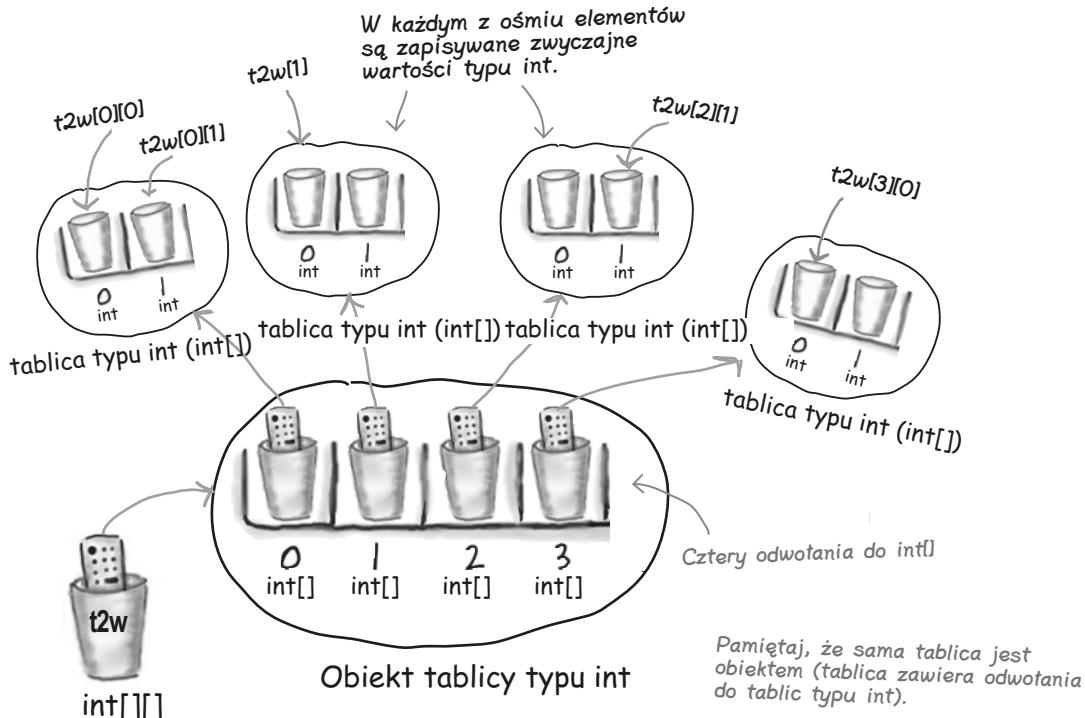
Tablice wielowymiarowe

Nr 2. Tablice wielowymiarowe

W większości języków programowania, jeśli stworzysz dwuwymiarową tablicę 4×2 , to mógłbyś ją sobie wyobrazić jako prostokąt o wymiarach 4 na 2, czyli zawierający w sumie 8 elementów. Jednak w Javie taka struktura danych składałaby się w rzeczywistości z 5 połączonych ze sobą tablic, bowiem w Javie tablica dwuwymiarowa jest po prostu *tablicą tablic!* (Analogicznie tablica trójwymiarowa będzie tablicą zawierającą tablice tablic). Oto jak działają tablice wielowymiarowe w Javie:

```
int[][] t2w = new int [4][2];
```

Wirtualna maszyna Java tworzy czteroelementową tablicę. Każdy z tych czterech elementów jest w rzeczywistości odwołaniem do nowo utworzonej, dwuelementowej tablicy typu int.



Stosowanie tablic wielowymiarowych

- Aby pobrać drugi element trzeciej tablicy: `int x = t2w[2][1]; // pamiętaj — indeksy zaczynają się od zera!`
- Aby utworzyć odwołanie do jednej z podtablic: `int[] kopia = t2w[1];`
- Uproszczony sposób inicjalizacji tablicy 2x3: `int[][] x = { { 2,3,4 }, { 7,8,9 } };`
- Aby stworzyć tablicę dwuwymiarową, której podtablice nie mają identycznej wielkości:

```
int[][] y = new int [2][]; // tworzymy tylko pierwszą dwuelementową tablicę  
y[0] = new int[3]; // tworzymy pierwszą podtablicę o długości 3 elementów  
y[1] = new int[5]; // tworzymy drugą podtablicę o długości 5 elementów
```

I w końcu najważniejsze zagadnienie, któremu prawie się udało znaleźć w tej książce...

Nr 1. Wyliczenia (nazywane także typami wyliczeniowymi)

Wspominaliśmy o stałych zdefiniowanych w API, takich jak na przykład `JFrame.EXIT_ON_CLOSE`. Możesz także tworzyć swoje własne stałe, oznaczając zmienne jako `static final`. Jednak czasami możesz chcieć określić zbiór wartości stałych, reprezentujący *jedynie* wartości, jakie może przyjmować zmienna. Taki zbiór dozwolonych wartości często jest określany terminem *wyliczenie*. Przed pojawiением się wersji 5.0 możliwości tworzenia wyliczeń w Javie były jedynie połowiczne. Jednak w Javie 5.0 zostały one znacznie rozbudowane i będą stanowiły powód zazdrości wszystkich Twoich przyjaciół używających jeszcze wcześniejszych wersji języka.

Kto należy do zespołu?

Załóżmy, że tworzysz witrynę WWW poświęconą Twójemu ulubionemu zespołowi muzycznemu i chciałbyś mieć absolutną pewność, że wszystkie komentarze będą adresowane do jego konkretnych członków.

Stary sposób tworzenia „wyliczenia”:

```
public static final int JAREK = 1;
public static final int JUREK = 2;
public static final int FILIP = 3;
```

// w dalszym miejscu, w kodzie...

```
if (wybranyCzlonekZespolu == JAREK) {
    // adresowane do JARKA
}
```

Mamy nadzieję, że w momencie dotarcia do tego fragmentu kodu zmienna `wybranyCzlonekZespolu` ma prawidłową wartość.

Takie rozwiązanie ma pewną zaletę — otóż UŁATWIA ono czytanie i analizę kodu. Dodatkowo, co także można uznać za zaletę, w takim przypadku nie można zmieniać wartości naszego udawanego wyliczenia — JAREK zawsze będzie mieć wartość 1. Niestety w razie stosowania tego rozwiązania nie ma żadnego prostego ani dobrego sposobu, by zagwarantować, że zmienna `wybranyCzlonekZespolu` przyjmie wyłącznie wartości 1, 2 lub 3. Jeśli jakiś głęboko ukryty fragment kodu przypisze jej wartość 812, to jest wysoce prawdopodobne, że w naszej aplikacji pojawią się jakieś problemy...

Wyliczenia

Nr 1. Wyliczenia — ciąg dalszy

A teraz ta sama sytuacja w przypadku wykorzystania wspaniałych wyliczeń udostępnianych w języku Java 5.0. Choć nasze wyliczenie jest bardzo proste, to jednak trzeba zauważać, że w większości przypadków wyliczenia są właśnie takie.

Nowe, oficjalne „wyliczenie”:

```
public enum Czlonkowie { JAREK, JUREK, FILIP };  
public Czlonkowie wybranyCzlonekZespolu;
```

// w dalszym miejscu kodu

```
if (wybranyCzlonekZespolu == Czlonkowie.JAREK) {  
    // adresowane do Jarka  
}
```

Już nie trzeba się przejmować wartością zmiennej!

To wygląda jak bardzo prosta definicja klasy, prawda? Okazuje się, że wyliczenia są specjalnym rodzajem klas. W tym przykładzie tworzymy nowy typ wyliczeniowy o nazwie „Czlonkowie”.

Zmienna „wybranyCzlonekZespolu” jest typu „Czlonkowie” i może przyjmować jedynie jedną z wartości „JAREK”, „JUREK” lub „FILIP”.

Składnia pozwalająca odwołać się do elementu „wyliczenia”.

Twoje wyliczenie dziedziczy po klasie `java.lang.Enum`

Tworząc nowe wyliczenie, tworzysz nową klasę, która *niejawnie dziedziczy po klasie `java.lang.Enum`*. Możesz zadeklarować wyliczenie jako niezależną klasę, umieścić ją w osobnym pliku źródłowym; możesz także zadeklarować ją jako składową innej klasy.

Stosowanie instrukcji `if` i `switch` z wartościami wyliczeniowymi

Typu wyliczeniowego można używać do tworzenia alternatywnych ścieżek wykonywania programu, wykorzystując do tego instrukcje `if` i `switch`. Warto także zauważać, że wartości typu wyliczeniowego można porównywać przy użyciu operatora `==` lub metody `equals()`. Zazwyczaj za „bardziej stylowe” uznaje się stosowanie operatora `==`.

```
Czlonkowie n = Czlonkowie.JUREK; ← Przypisujemy wartość wyliczeniową do zmiennej.  
if (n.equals(Czlonkowie.JAREK)) System.out.println("Jurek, Jurek!");  
if (n == Czlonkowie.JUREK) System.out.println("Jurek... czadu!");  
  
Czlonkowie cImie = Czlonkowie.FILIP;  
switch (cImie) {  
    case JAREK: System.out.print("zaśpiewaj ");  
    case FILIP: System.out.print("głośniej "); ← Szybki kwiz. Jakie będą wyświetcone wyniki?  
    case JUREK: System.out.print("grajku! ");  
}
```

Odpowiedź:

gotosnię, grajku!

Nr 1. Wyliczenia — ciąg dalszy

Naprawdę chytra wersja podobnego typu wyliczeniowego

Swój typ wyliczeniowy możesz uzupełnić o kilka dodatkowych elementów, takich jak konstruktor, metody, zmienne oraz coś, co jest określane jako „treść klasy” dla konkretnej wartości wyliczenia. Elementy te nie są powszechnie stosowane, jednak czasami można się na nie natknąć:

```
public class HfjEnum {
    enum Imiona {
        JAREK("gitara prowadząca") { public String spiewa() {
            return "tęsknie"; } },
        JUREK("gitara rytmiczna") { public String spiewa() {
            return "gardłowo"; } },
        FILIP("bas");
    }
    private String instrument;

    Imiona(String instrument) {
        this.instrument = instrument;
    }

    public String getInstrument() {
        return this.instrument;
    }

    public String spiewa() {
        return "od czasu do czasu";
    }
}

public static void main(String [] args) {
    for (Imiona n : Imiona.values()) {
        System.out.print(n);
        System.out.print(", instrument: " + n.getInstrument());
        System.out.println(", śpiewa: " + n.spiewa());
    }
}
```

To argument przekazywany w wywołaniu zadeklarowanego poniżej konstruktora.

To są tak zwane „treści klasy skojarzonej z konkretną wartością wyliczenia”. Wyobraź sobie, że przestanąją one podstawowe metody (w naszym przypadku jest to metoda `spiewa()`) typu wyliczeniowego. W tym przykładzie metody te zostaną wykonane w przypadku wywołania metody `spiewa()` dla wartości wyliczeniowej `JAREK` lub `JUREK`.

To jest konstruktor typu wyliczeniowego. Zostanie on wywołany jeden raz dla każdej zadeklarowanej wartości wyliczenia (czyli w naszym przykładzie zostanie on wykonany trzy razy).

Jak się zaraz przekonasz, te metody będą wywoływanie w metodzie `main()`.

Każdy typ wyliczeniowy posiada wbudowaną metodę `values()`, która zazwyczaj jest używana w pętlach `for`.

```
Wiersz polecenia
T:\>java HfjEnum
JAREK, instrument: gitara prowadząca, śpiewa: tęsknie
JUREK, instrument: gitara rytmiczna, śpiewa: gardłowo
FILIP, instrument: bas, śpiewa: od czasu do czasu
T:\>
```

Zauważ, że podstawowa wersja metody „`spiewa()`” jest wywoływanie wyłącznie w przypadku, gdy dla danej wartości typu wyliczeniowego nie ma skojarzonej z nią treści klasy.



Zagadka na pięć minut

Długa droga do domu

Kapitan Bajt dowodzący statkiem kosmicznym „Nieuustraszony przemierzacz gigabajtów” wchodzący w skład floty imperium Płaskonii otrzymał pilną transmisję z kwaterą głównej, opatrzoną sygnaturą „Ściśle tajne”. Wiadomość zawierała 30 doskonale zaszyfrowanych kodów nawigacyjnych, które

kapitan mógł użyć do wykreszenia bezpiecznego, choć prowadzącego przez sektory zajęte przez wroga, kursu do bazy. Wrogiem Płaskonii byli Hackarianie pochodzący z sąsiedniej galaktyki. Hackarianie wymyślili przerażający promień zniekształcający kody, który mógł stworzyć fałszywe obiekty na stercie jedynego komputera nawigacyjnego na pokładzie „Nieuustraszonego”. Co więcej, promień skanujący mógł także zmieniać poprawne

odwołania tak, aby wskazywały na te fałszywe obiekty. Jedyną obroną, jaką kapitan Bajt miał przeciw promieniowi skanującemu Hackarian, był skaner wirusów, który można było wbudować we wspaniałe oprogramowanie „Nieuustraszonego”.

W celu przetworzenia kluczowych kodów nawigacyjnych kapitan Bajt przekazał chorążemu Kowalskiemu następujące instrukcje programistyczne:

— Pierwszych pięć kodów umieść w tablicy KluczParsekow. Kolejnych 25 kodów podziel na pięcioelementowe grupy i umieść w dwuwymiarowej tablicy typu KluczKwadrantow. Oba te obiekty przekaź w wywołaniu metody `wykres1Kurs()` publicznej klasy finalnej Nawigacja. Po zwróceniu obiektu kursu uruchom skaner wirusów i sprawdź wszystkie zmienne referencyjne. Następnie uruchom program `NawSym` i przynieś mi wyniki.

Po kilku minutach chorąży wrócił z wynikami. — Wyniki programu `NawSym` gotowe do przejrzenia, kapitanie — zameldował. — Doskonale — odpowiedziała kapitan Bajt. — Chorąży, opisz wykonane czynności. — Tak jest, kapitanie! — odpowiedział Kowalski. — Najpierw zadeklarowałem i stworzyłem tablicę typu `KluczParsekow`, używając w tym celu instrukcji `KluczParsekow[] p = new KluczParsekow[5];`, następnie zadeklarowałem i stworzyłem tablicę typu `KluczKwadrantow`, używając w tym celu instrukcji `KluczKwadrantow[][] q = new KluczKwadrantow[5][5];`. Z kolei zapisałem pierwszych pięć kodów w tablicy `KluczParsekow`, używając przy tym pętli `for`, po czym zapisałem ostatnich 25 kodów w tablicy `KluczKwadrantow`, używając dwóch zagnieżdżonych pętli `for`. Następnie sprawdziłem wszystkie 32 odwołania przy użyciu skanera wirusów. Sprawdziłem jedno odwołanie `KluczParsekow`, pięć elementów tej tablicy, jedno odwołanie `KluczKwadrantow` oraz 25 elementów tej tablicy. Kiedy skaner poinformował o braku wirusów, uruchomiłem program `NawSym`, po czym, na wszelki wypadek, jeszcze raz uruchomiłem skaner.

Kapitan obrzucił Kowalskiego długim, lodowatym spojrzeniem i rzekł spokojnie: — Zostajesz skazany na areszt w swojej kwaterze za narażanie bezpieczeństwa tego statku i jego załogi, nie chcę cię więcej oglądać na mostku, dopóki nie nauczysz się dobrze Javy! Poruczniku Boolean, proszę przejąć funkcję chorążego i porządnie wykonać zadanie!

Dlaczego kapitan Bajt skazał chorążego na areszt?



Zagadka na pięć minut. Rozwiążanie



Długa droga do domu

Kapitan Bajt doskonale wiedział, że w Javie wielowymiarowe tablice są w rzeczywistości tablicami tablic. Tablica „q” typu KluczKwadrantów o wymiarach 5x5 wykorzystywałaby zatem 31 odwołań:

- 1 — zmienną referencyjną „q”;
- 5 — dla zmiennych referencyjnych $q[0]$ do $q[4]$;
- 25 — dla zmiennych referencyjnych $q[0][0]$ do $q[4][4]$.

Chorąży zapomniał o zmiennych referencyjnych zawierających odwołania do pięciu jednowymiarowych tablic „zagnieźdzonych” w tablicy „q”. Każde z tych pięciu odwołań mogło być przekształcone przez promień Hackerian, a testy wykonane przez Kowalskiego nigdy by tego nie wykazały.



Skorowidz

- , 141, 147
- &, 181, 682
- &&, 181
- (), 149
- ., 68, 86
- //, 42, 43
- ;, 42
- []], 596
- ^, 682
- _Skel, 640
- _Stub, 640
- {}, 685
- |, 181, 682
- ||, 181
- ~, 682
- +, 49, 321
- ++, 137, 141, 147
- <, 43
- <<, 682
- =, 43
- ==, 43, 44, 118, 584
- >, 43
- >>, 682
- >>>, 682

- A**
- abs(), 302, 314
- abstract, 228, 231, 253, 257
- accept(), 507
- ActionEvent, 385, 386, 394, 439
- ActionListener, 385, 386, 487, 688
- actionPerformed(), 385, 388, 394, 400
- add(), 162, 163, 167, 186, 236, 333, 334, 427
- addActionListener(), 385, 394, 688
- addKeyListener(), 388
- adres IP, 499, 501
- aktywność mózgu, 25
- algorytm mieszący, 587
- alternatywa, 181
- AND, 682
- animacja, 408, 410
- anonimowe klasy wewnętrzne, 688
- API, 28, 155
- aplety, 36, 626
- aplikacje biznesowe, 46
- aplikacje JWS, 621
- append(), 691
- architektura wielowarstwowa, 49
- architektura zastępcza Jini, 47, 654
- archiwum JAR, 73, 609, 619
 - lista zawartości, 617
 - manifest, 609, 616
 - pakiety, 616
 - plik manifestu, 617
 - pobieranie zawartości, 617
 - tworzenie, 609
 - uruchamianie, 610
- archiwum Javy, 73
- args, 40
- argumenty, 40, 106, 108, 110, 274
 - konstruktory klas bazowych, 283
 - polimorfizm, 593
 - przekazywanie, 108, 109
 - tablica obiektów, 592
 - zmienna lista argumentów, 328
- argumenty typu ogólnego, 565
- argumenty wiersza poleceń, 374
- ArrayList, 162, 163, 165, 166, 173, 185, 238, 315, 316, 554, 557, 582, 596, 611
 - add(), 163
 - contains(), 163
 - dodawanie elementów, 163
 - ilość elementów, 163
 - indexOf(), 163
 - isEmpty(), 163
 - położenie obiektu, 163

Skorowidz

ArrayList

remove(), 163
size(), 163
usuwanie elementów, 163

ArrayList<>, 167, 238, 317, 566

ArrayList<Object>, 239, 241, 243

asercje, 684

asin(), 314

assert(), 684

atrybuty, 62

autoboxing, 317

automatyczna konwersja, 317
argumenty, 318

operacje na liczbach, 319

przypisanie, 319

wartości wynikowe, 318

wyrażenia logiczne, 318

automatyczne odśmiecanie, 72

B

bezpieczeństwo, 51

bezpieczeństwo typów, 564

biblioteka Javy, 162

biblioteka kolekcji Javy, 553, 582

BigInteger, 327

BindException, 501

bit, 682

bit znaku, 682

bitowy operator alternatywy wykluczającej, 682

bitowy operator AND, 682

bitowy operator negacji, 682

bitowy operator OR, 682

blok kodu, 44, 685

pętle, 43

try-catch, 349

try-finally, 355

blokady, 535

wzajemna blokada wątków, 540

blokowanie tworzenia klas potomnych, 217

błędы, 156

boolean, 83

Boolean, 315

BorderLayout, 396, 429, 430, 438

regiony, 430

BoxLayout, 429, 437

break, 136, 137, 141

BufferedReader, 478, 483, 502, 509

BufferedWriter, 477, 483

bufory, 477

byte, 83

Byte, 315

C

Calendar, 331, 332, 333, 334

add(), 334

DATE, 334

DAY_OF_MONTH, 334

get(), 334

getInstance(), 332, 334

getTimeInMillis(), 334

HOUR, 334

HOUR_OF_DAY, 334

MILLISECOND, 334

MINUTE, 334

MONTH, 334

pola, 333, 334

roll(), 334

set(), 334

setTimeInMillis(), 334

YEAR, 334

ZONE_OFFSET, 334

catch, 349, 350

ceil(), 314

CGI, 647

char, 83

Character, 315

charAt(), 691

ClassCastException, 51, 244

CLASSPATH, 608, 615

close(), 456, 465

Collection, 582

Collection API, 582

Collections, 563, 575

sort(), 558, 559, 562

Color, 276
 Common Object Request Broker Architecture, 636
 Comparable, 573, 574, 576
 compareTo(), 573
 Comparator, 588, 590
 compare(), 580
 compare(), 580
 compareTo(), 573, 574, 576, 590
 Component, 230
 concat(), 691
 contains(), 162, 236
 CONTROLLER EVENT, 413
 ControllerEvent, 413, 416
 CORBA, 636
 cykl istnienia obiektów, 282, 286, 288, 292
 odśmiecanie, 288
 cykl życia zmiennej lokalnej, 286
 czas, 329
 czas istnienia obiektów, 286

D

dane globalne, 73
 dane MIDI, 345, 368
 data, 329
 Date, 329, 331
 DATE, 334
 daty, 329
 Calendar, 331, 332
 Date, 331
 GregorianCalendar, 331, 332
 inkrementacja, 333
 milisekundy, 333
 obsługa, 330
 ustawienia lokalne, 331
 znacznik czasu, 331
 DAY_OF_MONTH, 334
 default, 690
 definicja
 interfejsy, 252
 klasy, 42
 metody, 39

deklaracja, 87
 import statyczny, 335
 metody, 107
 składowe, 116
 wyjątki, 351, 363
 zmienne, 42, 71, 82, 83
 zmienne referencyjne, 87, 212, 268
 zmienne typów ogólnych, 565
 dekrementacja, 141, 147
 delete(), 686, 691
 deleteCharAt(), 686
 deserializacja, 304, 465, 466
 deserializacja kompozycji, 488
 diagramy, 29
 długość łańcucha, 691
 dodawanie komponentów 380, 427
 doGet(), 650
 dokumentacja API, 189, 190
 dokumentacja interfejsu programistycznego, 28
 dokumenty HTML, 650
 dokumenty JSP, 650
 domyślny poziom dostępu, 689
 doPost(), 650
 dostęp do składowych, 68
 dostęp chroniony, 690
 dostęp domyślny, 277
 dostęp prywatny, 218, 689
 dostęp publiczny, 218, 689
 double, 83
 Double, 315
 draw3DRect(), 392
 drawImage(), 389, 392, 394
 drawLine(), 392
 drawOval(), 392
 drawPolygon(), 392
 drawRect(), 392
 drzewo dziedziczenia, 204
 dynamiczna zmiana wielkości listy, 162, 165
 działanie, 103
 dziedziczenie, 63, 73, 194, 196, 209, 225
 drzewo dziedziczenia, 204

dziedziczenie

final, 217
hierarchia klas, 202
interfejsy, 252
klasy bazowe, 194, 278
klasy potomne, 194, 196, 211
poziomy dostępu, 208, 689
projektowanie z myślą o dziedziczeniu, 198
przesłanianie metod, 195, 196, 218
typy ogólne, 572
wywołanie metody, 203
wywołanie metody klasy bazowej, 208
zastosowanie, 201, 209, 210

dziedziczenie wielokrotne, 250, 257
śmiertelny romb, 251
dźwięki, 345

E

edytor tekstu, 28
EJB, 629, 647, 653
else, 45
Enterprise JavaBeans, 629, 647, 653
enum, 694
equals(), 118, 181, 237, 584, 585, 586, 587
Exception, 350, 358
extends, 236, 572, 598
extensions, 186
eXtreme Programming, 133

F

false, 43, 116
fałsz, 43
figury, 60, 389
File, 476, 483
FileInputStream, 465, 476, 483
FileOutputStream, 456, 457, 470, 471, 476, 483
FileReader, 478, 483
FileWriter, 471, 476, 477, 483
fill3DRect(), 392
fillOval(), 389, 391
fillRect(), 392, 411
fillRoundRect(), 392

filtry

, 483
final, 217, 310, 312
finally, 355
finalne, 311
float, 83
Float, 315
floor(), 314
FlowLayout, 429, 434, 436
for, 42, 43, 136, 137, 138, 146, 147
inicjalizacja, 146
kolekcje, 148
rozszerzone pętle for, 148
test logiczny, 146
wyrażenie iteracyjne, 146
format(), 323, 324
argumenty formatowania, 328
daty, 329
liczby, 323
łańcuch formatowania, 324
modyfikator typu, 327
formatowanie dat, 329
formatowanie liczb, 322
format(), 323, 324
instrukcje formatowania, 323
łańcuch formatowania, 324, 325
modyfikator typu, 327
spacje, 322, 324
specyfikator formatu, 326
Formatter, 322
FTP, 500, 501

G

generic method, 568
generics, 563
generowanie
 dźwięki, 368
 liczby losowe, 143
get, 111
GET, 643
get(), 162, 236, 334
getClass(), 237, 238
getContentPane(), 380, 389, 394, 396, 397, 426, 432

getInputStream(), 509
 getInstance(), 332, 334
 getOutputStream(), 509
 getSequencer(), 348
 getTimeInMillis(), 334
 gniazda, 495, 498, 509
 odczytywanie danych, 502
 ServerSocket, 507, 508, 509
 tworzenie, 502
 zapisywanie danych, 503
 godzina, 329
 gradient, 392, 393
 GradientPaint, 393
 graf obiektów, 460
 graficzny interfejs użytkownika, 60, 379, 380
 dodawanie komponentu do ramki, 380
 JButton, 380
 JFrame, 380
 JPanel, 390
 klasy anonimowe, 688
 kliknięcie przycisku, 382
 komponenty, 380
 menedżery układu, 427
 obsługa przycisku, 382
 okna, 380
 panel, 390
 przechwytywanie zdarzeń, 383
 przyciski, 380, 381
 rysowanie, 389, 390
 sposoby prezentacji, 382
 Swing, 381, 382, 425
 tworzenie, 380, 426
 układy, 396
 wyświetlanie elementów, 389
 wyświetlanie ramki, 380
 zdarzenia, 382
 grafika, 390, 394
 Graphics, 390, 391, 392
 Graphics2D, 392, 394
 GregorianCalendar, 331, 332
 GUI, 36, 396

H

hashCode(), 237, 241, 584, 585, 586, 587
 hashing algorithm, 587
 HashMap, 315, 557, 582
 HashSet, 557, 582, 583, 585, 587
 equals(), 585
 hashCode(), 585
 Hashtable, 582
 hermetyzacja, 93, 112, 114
 hierarchia dziedziczenia, 207
 hierarchia klas, 202, 226
 historia Javy, 36
 HOUR_OF_DAY, 334
 HTML, 650
 HTTP, 500, 501, 643
 HttpServlet, 648, 650

I

IDE, 684
 identyfikator wersji, 484
 if, 42, 45, 136, 694
 if-else, 42, 45
 IIOP, 636
 ImageIcon, 394
 implementacja
 interfejsy, 252, 253
 metody, 133
 metody abstrakcyjne, 232
 implements, 252
 import, 187
 import statyczny, 335
 indexOf(), 162, 186, 236
 inicjalizacja obiektu, 269, 271, 272
 inicjalizacja składowych, 116
 składowe statyczne, 309
 inicjalizator statyczny, 310
 inkrementacja, 137, 141, 147
 InputStreamReader, 502, 509
 insert(), 686, 691
 instanceof, 244
 instrukcje, 39, 42
 instrumenty MIDI, 373

int, 83, 316
Integer, 137, 149, 315, 316, 683
Integer.parseInt(), 137, 138, 141, 149
interface, 252
interfejsy, 252, 255, 257
 ActionListener, 386, 688
 Collection, 582
 Comparable, 573, 574, 576
 definicja, 252
 implementacja, 252, 253
List, 581
Map, 581
 metody, 252, 253
MouseListener, 384
polimorfizm, 253, 254
Remote, 638
Runnable, 254, 516, 518, 524
Serializable, 254, 461, 638
Set, 581
 tworzenie, 253
intreface, 252
IOException, 471
isEmpty(), 162, 186
itemStateChanged(), 388, 442

J

J2EE, 36
J2ME, 36
J2SE, 36, 345
jar, 73, 609, 617
JAR, 606, 609, 619
java, 186, 619
Java, 34
Java 1.02, 36
Java 1.1, 36
Java 2, 36
Java 2 Standard Edition SDK, 28
Java 5.0, 36
Java API, 155, 165, 184
 dokumentacja, 189
Java Collections Framework, 553
Java Enterprise Edition, 650, 653

Java I/O API, 457
Java Network Launch Protocol, 623
Java Reflection API, 304
Java Server Pages, 650
Java Standard Edition, 162
Java Virtual Machine, 35
Java Web Start, 606, 621, 622, 626
 aplikacja pomocnicza, 622
 ograniczenia bezpieczeństwa, 626
 plik .jnlp, 623
 przeglądarka WWW, 622
 serwer WWW, 622
 wdrażanie aplikacji, 624
java.awt, 186
java.awt.event, 384
java.io, 186, 483
java.lang, 184, 186, 315, 513, 514
java.lang.Thread, 514
java.net, 186, 495
java.net.Socket, 499
java.nio, 483
java.rmi.Remote, 638
java.rmi.server, 639
java.text, 322
java.util, 184, 185, 186, 322
java.util.Calendar, 331
java.util.Collections, 563
java.util.Date, 331
java.util.GregorianCalendar, 331, 332
Java2D API, 389
javac, 28, 572, 608
javac -d, 614, 615
JavaSound API, 345
javax, 186
javax.sound.midi, 346, 374
javax.swing, 184, 186
jawne przypisanie odwołaniu wartości null, 291
jawne rzutowanie typów, 110
JButton, 380, 426, 427
JCheckBox, 380, 442
JComponent, 426
JEE, 647, 650, 653

JEST, 205, 206, 207, 209
 język Java, 34
 język zorientowany obiektowo, 44
 JFrame, 380, 394, 426, 438
 Jini, 47, 654, 658
 dzierżawa, 657
 klient, 654
 odkrywanie adaptacyjne, 655, 656
 sieci samoaktualizujące, 657
 wyszukiwanie usługi, 654
 JLabel, 380
 JList, 380, 443
 JPanel, 390, 426
 JPEG, 389
 JRadioButton, 380
 JRMP, 636
 JScrollPane, 380, 440
 JSlider, 380
 JSP, 650
 JTable, 380
 JTextArea, 380, 440, 441
 JTextField, 380, 439
 JVM, 35, 41, 50, 64, 72, 264, 610
 JWS, 621

K

kalendarz, 331, 334
 Calendar, 332
 pobieranie, 332
 KeyEvent, 388
 keyTyped(), 388
 klasy, 39, 40, 44, 63, 67, 73, 82, 141, 255
 ActionEvent, 386
 ArrayList, 162, 165, 557
 BorderLayout, 429, 430
 BoxLayout, 429, 437
 BufferedReader, 478, 502
 BufferedWriter, 477
 Calendar, 331, 332
 Collections, 563
 Color, 276
 Component, 230

Date, 329, 331
 dziedziczenie, 63, 194, 210
 Exception, 350
 File, 476
 FileInputStream, 465
 FileOutputStream, 456, 457
 FileReader, 478
 FileWriter, 477
 FlowLayout, 429, 434
 Formatter, 322
 Graphics, 390, 391
 Graphics2D, 392
 GregorianCalendar, 331, 332
 HashMap, 557
 HashSet, 557
 hermetyzacja, 112
 hierarchia klas, 202
 HttpServlet, 648
 implementacja interfejsu, 254
 InputStreamReader, 502
 Integer, 149
 interfejsy, 254
 JCheckBox, 442
 JComponent, 426
 JFrame, 380, 426
 JPanel, 390, 426
 JTextArea, 440, 441
 JTextField, 439
 klasy finalne, 217, 311, 312
 klasy konkretne, 228, 230
 klasy potomne, 63, 194, 195, 196, 216, 255
 klasy użytkowe, 184
 klasy wewnętrzne, 402
 klasy zewnętrzne, 402, 403
 konstruktory, 269
 LinkedHashSet, 557
 LinkedList, 557, 558
 Math, 184, 302
 metody, 39, 66
 metody statyczne, 303
 modyfikatory dostępu, 113
 MouseEvent, 384

- klasy
 - NIO, 483
 - Object, 236, 237
 - ObjectOutputStream, 456, 457
 - organizowanie klas, 608
 - pakiety, 185, 186, 611
 - pełne nazwy, 187
 - poziomy dostępu, 208
 - PrintWriter, 503
 - projektowanie, 66
 - RuntimeException, 352
 - Sequencer, 346
 - ServerSocket, 507
 - składowe, 66, 116
 - składowe statyczne, 303, 307
 - Socket, 499
 - String, 184, 691
 - StringBuffer, 691
 - System, 184
 - Thread, 513, 514, 524
 - TreeSet, 557, 558, 588
 - tworzenie, 131
 - typy podstawowe, 315
 - umieszczanie w pakiecie, 613
 - UnicastRemoteObject, 639
- klasy abstrakcyjne, 228, 230, 238, 255
 - metody abstrakcyjne, 231
 - tworzenie, 228
- klasy bazowe, 63, 73, 194, 236, 277, 278
 - konstruktory, 279
 - składowe, 278
 - wywołanie konstruktora, 281
- klasy uogólnione, 566
 - ArrayList<>, 566
 - metody uogólnione, 568
 - parametry typu, 567
- klasy wewnętrzne, 217, 379, 402, 403, 406, 688
 - animacja, 408
 - anonimowe klasy, 688
 - obiekty, 404
 - tworzenie, 402
- klasy zagnieżdżone, 687
- anonimowe klasy, 687, 688
- statyczne klasy, 687
- klient, 497, 504, 510
- klient pogawędka, 510
- kliknięcie przycisku, 382
- klucze, 591
- klucz-wartość, 581
- kod bajtowy, 34, 50
- kod HTML, 650
- kod metody, 39
- kod pomocniczy, 135
- kod przygotowawczy, 131, 136
- kod testowy, 131, 133, 134, 135
- kod właściwy, 131
- kod wynikowy, 34, 35
- kod źródłowy, 34, 35
- kolekcje, 148, 553
 - ArrayList, 557, 582, 596
 - biblioteka kolekcji, 582
 - Collection, 582
 - Collection API, 582
 - equals(), 585
 - hashCode(), 585
 - HashMap, 557, 582
 - HashSet, 557, 582, 583, 585
 - Hashtable, 582
 - LinkedHashMap, 582
 - LinkedHashSet, 557, 582
 - LinkedList, 557, 558, 582
 - List, 581
 - listy, 581
 - Map, 581, 582
 - mapy, 581, 591
 - Set, 581
 - sortowanie, 558
 - TreeMap, 582
 - TreeSet, 557, 558, 582, 588
 - typy ogólne, 564
 - Vector, 582
 - zbiory, 581
- kolizje nazw, 185
- kolory, 391, 397

- komentarze, 42
 komparatory, 576, 577
 komplikacja, 614
 opcja -d, 608, 614
 kompilator, 34, 35, 50
 komponenty, 380
 JCheckBox, 442
 JComponent, 426
 JFrame, 426, 438
 JPanel, 426
 JScrollPane, 440
 JTextArea, 440, 441
 JTextField, 439
 komponenty interakcyjne, 426
 komponenty tła, 426
 pasek przewijania, 440
 pole tekstowe, 439
 pole wyboru, 442
 Swing, 426
 wielowierszowe pole tekstowe, 440
 wymiary, 438
 komponenty EJB, 629, 647
 komunikaty MIDI, 371, 372, 373
 konfiguracja Javy, 28
 konflikty nazw, 611
 koniec linii dziedziczenia, 217
 koniunkcja, 181
 konkatenacja, 49, 321
 konkretne klasy potomne, 230
 konstruktor, 269, 271, 277
 argumenty, 274
 dostęp domyślny, 277
 inicjalizacja obiektu, 272
 konstruktor bezargumentowy, 273, 274, 276
 łańcuchowe wywołanie konstruktorów, 279
 modyfikatory dostępu, 277
 przeciążenie, 275
 super(), 281, 284
 this(), 284
 wiele konstruktorów, 273
 wywołanie, 269
 wywołanie konstruktora klasy bazowej, 281
 wywołanie przeciążonych konstruktorów, 284
 zastosowanie, 270
 konstruktorы klas bazowych, 278, 279
 argumenty, 283
 super(), 281
 wywołanie, 281, 282
 kontrakt, 211, 218
 kontrola dostępu, 217
 kontrola typów, 238
 kontrola wersji obiektów, 484
 konwersja liczby na łańcuch znaków, 321
 konwersja łańcucha znaków na liczbę, 136, 137, 149
 konwersja łańcucha znaków na wartość typu
 podstawowego, 320
 kropka, 68, 86
 kubełek, 587
- L**
- length, 186
 length(), 691
 liczby, 83, 301
 formatowanie, 322
 liczby całkowite, 44, 83, 327
 liczby losowe, 49, 143, 314
 liczby szesnastkowe, 327
 liczby zmiennoprzecinkowe, 83, 327
 Math, 302
 zamiana na łańcuch znaków, 321
 licznik pętli, 146
 LinkedHashMap, 582
 LinkedHashSet, 557, 582
 LinkedList, 557, 558, 582
 List, 581, 582
 lista argumentów, 117
 lista zawartości archiwum JAR, 617
 listy, 443, 554, 581
 sortowanie, 554
 literaly, 84
 lokalne aplikacje Javy, 610
 long, 83
 Long, 315
 lookup(), 642

Ł

łańcuch formatowania, 324
argumenty formatowania, 328
składnia, 325
specyfikator formatu, 326
łańcuchowe wywołanie konstruktorów, 279
łańcuchy znaków, 38, 40, 45, 81, 134, 143, 691
konkatenacja, 49, 321, 691
konwersja na wartość typu podstawowego, 320
łączenie, 49, 321, 691
niezmienność, 683
przetwarzanie, 482
String, 691
StringBuffer, 691
zapisywane w pliku tekstowym, 471
łączenie łańcuchów znaków, 49, 321, 691

M

MA, 205, 209, 270
main(), 40, 41, 44, 70, 73, 140, 142, 304, 515
maksimum, 314
manifest, 73, 609, 616
MANIFEST.MF, 617
manipulowanie datami, 330
Map, 581, 582
mapy, 581, 591
maszyna wirtualna, 34
Math, 184, 302, 314
 abs(), 302, 314
 max(), 302, 314
 min(), 302, 314
 PI, 310
 random(), 143, 305, 314
 round(), 314
max(), 302, 314
mechanizm automatycznej konwersji, 317
mechanizm rozsyłania grupowego IP, 654
mechanizm zarządzający wątkami, 521
menedżery układu, 425, 427, 428, 438, 442
 BorderLayout, 429, 430
 BoxLayout, 429, 437
 dodawanie komponentów, 427

FlowLayout, 429, 434, 436
scenariusz określania układu, 428
wyłączanie, 438
zasady rozmieszczenia, 428
Message, 371
META-INF, 617
Metal, 382
metapoznanie, 25, 65
metody, 39, 63, 64, 66, 73, 103, 104, 110, 196, 303
 add(), 236
 argumenty, 106, 108, 110
 contains(), 236
 deklaracja, 107
 dziedziczenie, 203
 equals(), 118, 237, 584, 586, 587
 format(), 324
 get, 111
 getClass(), 237, 238
 hashCode(), 237, 584, 586, 587
 implementacja, 133
 indexOf(), 236
 interfejsy, 253
 konstruktory, 269
 lista argumentów, 117
 main(), 40, 41, 44, 70, 140, 142, 304
 metody finalne, 217, 311, 312
 metody globalne, 73
 metody klasy bazowej, 208
 metody publiczne, 73, 113
 metody uogólnione, 568
 metody ustawiające, 111
 metody zwracające, 111
 parametry, 106, 111
 poziomy dostępu, 218
 printf(), 322
 przeciążanie, 219
 przesłanianie, 195, 200, 218
 return, 107
 set, 111
 sort(), 556
 split(), 482
 stos wywołań, 265

- synchronized, 534
- toString(), 237, 321
- ustawianie danych, 111
- void, 107
- wartości wynikowe, 110, 111
- wyjątki, 351, 353
- wywołanie, 104, 108, 203, 245
- wywołanie metody zdefiniowanej w klasie bazowej, 256
- zawartość, 39
- zmienna lista argumentów, 328
- zwracanie danych, 107, 111
- metody abstrakcyjne, 231
 - definicja, 231
 - implementacja, 232
 - polimorfizm, 231
 - zastosowanie, 231
- metody obsługi zdarzeń, 388
- metody statyczne, 73, 302, 303, 304, 306, 312, 332
 - tworzenie, 303
 - wywołanie, 304
 - zasada działania, 305
- MIDI, 344, 345
 - dane piosenki, 371
 - długość nuty, 373
 - generowanie dźwięków, 368
 - instrukcje, 371
 - instrumenty, 373
 - kanały, 372
 - komunikaty, 371, 372, 373
 - MidiEvent, 369
 - NOTE OFF, 372
 - NOTE ON, 372
 - sekvenser, 368, 369
 - syntezator, 368
 - zdarzenia, 368, 371, 375
 - zmiana długości nuty, 373
 - zmiana instrumentu, 373
 - zmiana komunikatu, 373
 - zmiana nuty, 373
- MidiEvent, 369, 371
- MidiSystem.getSequencer(), 366
- MidiUnavailableException, 349, 366
- milisekundy, 333
- MILLISECOND, 334
- MIME, 624
- min(), 302, 314
- minimum, 314
- MINUTE, 334
- modyfikacja drzewa klas, 246
- modyfikacja klas, 248
- modyfikatory dostępu, 113, 689
 - private, 689
 - protected, 689, 690
 - public, 689
- MONTH, 334
- MouseEvent, 384
- mouseExited(), 388
- MouseListener, 384
- mouseMoved(), 384
- mousePressed(), 384, 388
- mouseReleased(), 384
- mózg, 23
- MuzMachina, 344, 672
- muzyka MIDI, 344, 346
- mysz, 384
- myślenie, 23

N

- namiastka RMI, 636, 640
- Naming.lookup(), 642
- Naming.rebind(), 644, 645
- naruszenia typów danych, 51
- nauczanie, 24
- nawiązywanie połączenia sieciowego, 498, 499
- nazwy, 42, 185
 - pakiety, 612
 - zmienne, 82, 85, 310
- Network Launch Protocol, 623
- new, 72, 87, 268, 269, 270, 275, 279
- niestatyczne zmienne finalne, 311
- niezależny wątek realizacji, 524
- niezmiennosć, 683
- NIO, 483
- niszczenie obiektów, 288

NOT, 682
NOTE OFF, 372, 413
NOTE ON, 372, 413
null, 94, 116, 291
NullPointerException, 352
NumberFormatException, 320
numer portu TCP, 499

O

obiekt zdarzenia, 384
obiekty, 59, 60, 66, 67, 129, 242
 blokady, 535
 cykl istnienia, 282, 286
 deserializacja, 465, 466
 dostęp do składowych, 68
 działanie, 66, 103, 104
 hermetyzacja, 112
 inicjalizacja, 116, 269, 271, 272
 klasy wewnętrzne, 404
 konstruktory, 269
 kontrola wersji obiektów, 484
 metody, 66
 modyfikatory dostępu, 113
 niszczenie, 288
 odśmietanie, 288
 odtwarzanie obiektów, 465
 odwołanie do obiektu, 87, 88
 przechowywanie danych, 264
 równość obiektów, 118, 584
 serializacja, 454, 458
 składowe, 66, 116
 składowe statyczne, 307
 stan, 66, 103, 104, 459
 sterta, 89
 tworzenie, 68, 87, 268, 280
 zapisywanie, 454
obiekty pomocnicze RMI, 636
Object, 236, 237, 238, 257
 equals(), 237
 getClass(), 237
 hashCode(), 237
 toString(), 237

ObjectInputStream, 483
ObjectOutputStream, 456, 457, 470, 471, 483
obliczenia matematyczne, 301
obrazy, 389
 JPEG, 389
obsługa dat, 330
obsługa dźwięku, 345
obsługa plików tekstowych, 472
obsługa sytuacji wyjątkowych, 352
obsługa wejścia-wyjścia, 457, 483
obsługa wyjątków, 343
obsługa zdarzeń, 379, 383, 388, 414, 415
obszar łańcuchów znaków, 683
odbieranie danych, 498
odbieranie zdarzeń, 416
 zdarzenie niezwiązane z interfejsem użytkownika, 413
odbiorca zdarzeń, 384, 385, 387
odczytywanie danych
 gniazda, 502
 pliki, 465, 478
oddzielanie kodu źródłowego od plików klasowych, 608
odkrywanie adaptacyjne, 654, 655, 656
odśmiecacz sterty, 263, 288
odśmiecana sterta, 90
odśmietanie, 288
odświeżenie komponentu, 390
odtwarzanie kompozycji, 488
odtwarzanie muzyki MIDI, 344, 346
odtwarzanie obiektów, 465
odwołanie do obiektu, 82, 86, 87, 88, 94, 244
odwrotne dziedziczenie, 208
odwrócone nazwy domen, 612
okna, 380
opakowanie typów, 315
operacje atomowe, 534
operacje bitowe, 682
operacje na datach, 330, 331
operacje na liczbach, 319
operacje wejścia-wyjścia, 457, 477
 bufory, 477
operatory
 ==, 118, 584

- alternatywa, 181
 bitowe, 181
 dekrementacja, 141, 147
 inkrementacja, 141, 147
 instanceof, 244
 koniunkcja, 181
 kropka, 68
 new, 268, 269, 275, 279
 porównania, 43
 przesunięcie bitowe, 682
 przetwarzanie skrócone, 181
 przypisanie, 43
 rzutowanie typów, 149
 opróżnienie bufora, 477
 OR, 682
 organizowanie klas, 608, 618
 oszczędzanie pamięci, 683
- P**
- pack(), 438
 package, 613
 paintComponent(), 390, 391, 392, 395, 409
 pakiety, 185, 186, 611
 archiwum JAR, 616
 komplikacja, 614
 konflikty nazw, 611
 konwencja nazewnicza, 612
 MANIFEST.MF, 617
 META-INF, 617
 nazwy, 612
 organizowanie klas, 618
 przestrzenie nazw, 185
 uruchamianie programu, 614
 zapobieganie konfliktom nazw, 612
- panel, 390, 438
 parametry, 106, 111, 117
 parametry typu, 567, 568
 parseBoolean(), 320
 parseDouble(), 320
 parseInt(), 136, 138, 149
 pasek przewijania, 440
 PATH, 28
- pełne nazwy klas, 187
 pętla stanów wątku, 520
 pętle, 42, 43, 147
 blok, 43
 break, 137
 for, 43, 137, 146, 147
 licznik, 146
 rozszerzone pętle for, 148
 test warunku, 43
 while, 43, 44, 147
 zakończenie wykonywania, 137
 pisanie kodu testowego, 133, 134, 135
 plik manifestu, 617
 plik źródłowy, 39
 pliki, 476
 .class, 41
 .jar, 73
 .java, 39
 .jnlp, 623
 File, 476
 MIDI, 345
 odczytywanie, 465, 478
 zapisywanie, 456, 471, 476
 pliki klasowe, 609
 pliki klasowe RMI, 644
 pliki tekstowe, 454, 472, 483
 odczytywanie, 478
 zapisywanie, 472
 płótno, 390
 pobieranie
 dane wejściowe, 143
 kalendarze, 332
 zawartość archiwum JAR, 617
 znacznik czasu, 331
 poczta elektroniczna, 500
 podstawowa biblioteka Javy, 162
 podstawowe typy danych, 83
 pola, 333
 pole tekstowe, 439
 pole wyboru, 442
 polimorfizm, 193, 211, 212, 214, 218, 225, 234, 243
 interfejsy, 253, 254

- polimorfizm
 - metody abstrakcyjne, 231
 - Object, 239
 - stosowanie, 216
 - typy ogólne, 594
- połączenia sieciowe, 495, 498
 - accept(), 507
 - adres IP, 499
 - BindException, 501
 - BufferedReader, 502, 509
 - gniazda, 498, 509
 - InputStreamReader, 502, 509
 - klient, 497, 504, 510
 - nawiązywanie połączenia, 498, 499
 - odbieranie danych, 498, 512
 - odczytywanie danych z gniazda, 502
 - porty TCP, 499
 - powszechnie znane porty, 500
 - PrintWriter, 503, 509
 - ServerSocket, 507, 508, 509
 - serwer, 497, 498, 507, 509
 - Socket, 499
 - strumienie, 509
 - wielowątkowość, 513
 - wysyłanie danych, 498, 512
 - zapisywanie danych w gnieździe, 503
- połączenie z serwetlem, 649
- połączone wywołania, 686
- pomocnicy, 634
 - pomocnik klienta, 636
 - pomocnik serwera, 636
- POP3, 500
- poprawianie błędu, 158
- porównania, 43
 - odwołania do obiektów, 118
 - typy podstawowe, 118
 - zmienne, 118
- Portal, 180
- porty TCP, 499
 - powszechnie znane porty, 500
- postdekrementacja, 141
- postinkrementacja, 137, 141
- pośrednik RMI, 636
- powielanie kodu, 199, 210
- powszechnie znane porty, 500
- powtórzenia, 580
- poziomy dostępu, 208, 218, 689
 - chroniony, 689, 690
 - domyślny poziom dostępu, 689
 - prywatny, 689
 - publiczny, 689
- prawda, 43
- predekrementacja, 141
- preinkrementacja, 141
- prezentacja grafiki, 390
- print(), 45, 503, 509
- printf(), 322
- println(), 45, 503, 509
- PrintWriter, 503, 509, 511
- priorytety wątków, 546
- private, 113, 277, 304, 689
- procedury, 129
- program, 38, 41, 73, 606
- programowanie ekstremalne, 133
- programowanie obiektowe, 63, 65
- projekt gry, 129
- projekt wywoływanie zdalnych metod, 633
- projektowanie, 129
 - projektowanie drzewa dziedziczenia, 204
 - projektowanie klas, 66
 - projektowanie z myślą o dziedziczeniu, 198
- prostokąty, 389
- protected, 689, 690
- protokoły, 501
- protokół uruchamiania przez sieć aplikacji Javy, 623
- prywatne konstruktory, 304
- prywatne metody, 304
- przechowywanie informacji, 264, 553
 - metody, 265
 - obiekty, 264
 - odwołania do obiektów, 266
 - składowe, 264
 - zmienne lokalne, 264, 267

przechwytywanie wyjątków, 350
 przechwytywanie wielu wyjątków, 357
 przechwytywanie zdarzeń, 383, 386, 399
 przeciążanie metod, 219
 konstruktory, 275
 Przeglądarka usług uniwersalnych, 658, 660
 przeglądarka WWW, 500
 przekazywanie argumentów, 106, 108
 przez wartość, 109, 119
 przesłanianie metod, 64, 195, 196, 200, 218
 przestrzenie nazw, 185
 przesunięcie bitowe, 682
 przetwarzanie łańcuchów znaków, 482
 przyciski, 380, 381, 435
 obsługa, 382
 przypisanie, 43, 44, 83, 84, 87, 319
 zmienne typów ogólnych, 565
 pseudokod, 131, 132
 public, 73, 85, 113, 217, 253, 277, 310, 689
 punkt startowy programu, 40, 41

R

ramka stosu, 265, 287
 random(), 49, 73, 143, 305
 readLine(), 478
 readObject(), 465, 470
 rebind(), 644
 referencje, 86
 refleksja, 304
 reguły przesłaniania, 218
 reguły stosowania wyjątków, 366
 rejestr RMI, 637, 639, 645, 658
 rejestracja odbiorcy, 416
 rejestracja usługi w rejestrze RMI, 639
 relacje
 JEST, 205, 206, 207, 209
 MA, 205, 209, 270
 Remote, 638, 661
 Remote Method Invocation, 467, 629, 632, 645
 RemoteException, 638, 639, 645
 remove(), 162, 186
 repaint(), 390, 395, 409

replace(), 691
 reprezentacja wartości typów podstawowych
 w formie obiektów, 315
 requestFocus(), 439
 return, 107, 355
 reverse(), 691
 rezygnowanie z obsługi wyjątku, 364
 RGB, 276
 RMI, 467, 606, 629, 632, 645, 650, 658
 generowanie namiastki, 640
 generowanie szkieletu, 640
 HTTP, 643
 IIOP, 636
 implementacja zdalnego interfejsu, 639
 JRMP, 636
 klient, 642
 namiastka, 636, 640
 obiekty pomocnicze, 636
 pliki klasowe, 644
 pomocnicy, 634, 637
 pomocnik klienta, 636
 pomocnik serwera, 636
 pośrednik, 636
 protokoły, 636
 rejestr, 637, 639
 rejestracja usługi w rejestrze RMI, 639
 Remote, 638
 RemoteException, 639
 serwer, 641
 szkielet, 636, 640
 tworzenie zdalnego interfejsu, 638
 tworzenie zdalnej usługi, 637
 UnicastRemoteObject, 639
 uruchamianie programu rmiregistry, 640
 uruchamianie usługi, 640
 wyjątki, 638
 wywołanie metody, 635
 zastosowanie, 646
 zdalny interfejs, 638
 zdobywanie klasy pośrednika, 643
 zdobywanie namiastki, 642
 rmiregistry, 637, 640, 644

Skorowidz

- roll(), 333, 334
- rotate(), 392
- round(), 314
- rozgałęzienia warunkowe, 42, 45
- rozpoznanie aplikacji, 605, 606
- rozszerszanie klasy bazowej, 196
- rozszerszenia, 186
- rozszerszone pętle for, 148
- równość, 44
 - obiekty, 118, 584
 - odwołania, 584
- run(), 516, 517, 522, 530
- Runnable, 254, 516, 517, 518, 524, 536
 - run(), 516
- RuntimeException, 352
- rysowanie, 389, 390, 395, 417
 - gradient, 392, 393
 - GradientPaint, 393
 - Graphics, 390, 391, 392
 - Graphics2D, 392
 - kolory, 391
 - koło wypełnione losowo wybranym kolorem, 391
 - odświeżenie komponentu, 390
 - paintComponent(), 390, 391
 - płótno, 390
 - repaint(), 390, 395
 - wyświetlanie plików JPEG, 391
 - zdarzenia, 395
- rzutowanie typów, 110, 143, 149, 244, 465
- S**
 - Sampled, 345
 - scale(), 392
 - sekvenser, 346, 369
 - selectAll(), 439
 - Sequence, 369
 - Sequencer, 346, 366, 369
 - Serializable, 254, 461, 464, 470, 638
 - serializacja, 454, 455, 457, 458, 462, 464, 468, 470
 - graf obiektów, 460
 - kontrola wersji obiektów, 484
 - pomijanie składowych, 463
- Serializable, 461
- składowe statyczne, 467
- transient, 463
- serializacja kompozycji, 487
- serialVersionUID, 485
- ServerSocket, 507, 508, 509
- servlets.jar, 648, 650
- serwer, 497, 498, 501, 507, 509
 - accept(), 507
 - komunikacja z klientem, 507
 - tworzenie, 507
 - wielowątkowość, 513
- serwer aplikacji korporacyjnych, 653
- serwer czasu, 500
- serwer EJB, 653
- serwer poczty elektronicznej, 500
- serwer pogawędka, 500
- serwer WWW, 500, 647
- serwlety, 606, 647, 649, 650, 651
 - HttpServlet, 648, 650
 - klasa serwletu, 648
 - położenie plików klasowych, 648
 - servlets.jar, 648, 650
 - strona WWW, 648
 - tworzenie, 648
 - uruchamianie, 648
- set, 111
- Set, 581, 582
- set(), 333, 334
- setBackground(), 435, 437
- setCharAt(), 691
- setColor(), 391, 392, 394, 411
- setContentPane(), 438
- setDefaultCloseOperation(), 397
- setFont(), 432
- setHorizontalScrollBarPolicy(), 440
- setLayout(), 437, 438
- setMessage(), 371, 372
- setRenderingHints(), 392
- setSelected(), 442
- setSize(), 380, 394, 426, 438
- setText(), 383, 439

- setTime(), 333
- setTimeInMillis(), 333, 334
- setVerticalScrollBarPolicy(), 440
- setVisible(), 380, 394, 426
- shear(), 392
- short, 83
- Short, 315
- sieci samoaktualizujące, 654, 657
- sieć, 499
 - sieć domowa, 500
- size(), 162, 186
- składnia, 42
- składowe, 66, 73, 103, 104, 110, 116, 117, 196, 264, 336
 - cykl istnienia, 292
 - deklaracja, 116
 - inicjalizacja, 116
 - prywatne, 113
 - publiczne, 113
 - transient, 463
 - wartości domyślne, 116
- składowe statyczne, 303, 307, 308, 312, 336
 - inicjalizacja, 309
- skrócone przetwarzanie operatorów, 181
- skrypty CGI, 647
- sleep(), 520, 521, 525, 526, 541
- słowa kluczowe, 85
- słowa zarezerwowane, 85
- SMTP, 500, 501
- Socket, 499, 502, 507, 509, 511, 611
- Software Development Kit, 28
- sort(), 556, 558, 559, 563, 569, 571, 575, 590
- SortedMap, 582
- SortedSet, 582
- sortowanie, 570
 - listy, 554
 - TreeSet, 590
 - własne obiekty, 560
- sparametryzowane typy, 167
- specyfikacja, 60, 62
- specyfikator formatu, 326
- split(), 482
- sprawdzanie klasy obiektu, 244
- sqrt(), 314
- stałe, 310
- stan obiektu, 66, 104, 110, 459
 - zapisywanie, 454
- stan wątku, 519
 - pętla stanów, 520
 - wątek realizacji, 519
 - wątek uruchamialny, 519
 - wątek zablokowany, 520
- start(), 516, 522, 524
- static, 73, 85, 305, 306, 310
- statyczne klasy zagnieżdżone, 687
- statyczne zmienne finalne, 310
 - inicjalizator statyczny, 310
 - określanie wartości, 310
- sterowanie przepływem w blokach try-catch, 354
- sterta, 72, 89, 264
- sterta automatycznie odśmiecana, 72
- stos, 264, 265, 287
 - odwołania do obiektów, 266
 - ramka stosu, 265
 - wierzchołek, 265
 - zmienne lokalne, 267
- stos wywołań, 265, 515
- stosowanie polimorficznych odwołań typu Object, 239
- stosowanie wątków, 528
- stosowanie wielu odbiorców zdarzeń, 400
- String, 38, 40, 49, 81, 143, 184, 217, 320, 321, 471, 683, 691
 - metody, 691
 - split(), 482
- StringBuffer, 686, 691
 - metody, 691
- strony JSP, 650
- struktura kodu, 39
- strukturny danych, 91, 553
- strumienie, 457, 470, 483, 509
 - BufferedReader, 502, 509
 - BufferedWriter, 477, 483
 - efektywność odczytu, 483
 - InputStream, 465, 476, 483
 - OutputStream, 456, 457, 483
 - Writer, 471, 476, 477, 483

Skorowidz

strumienie

- filtry, 483
- gniazda, 502
- InputStreamReader, 502, 509
- ObjectInputStream, 483
- ObjectOutputStream, 456, 457, 483
- PrintWriter, 503, 509
- strumienie łańcuchowe, 457
- strumienie połączeniowe, 457
 - zamykanie, 456
- substring(), 691
- super, 208, 256
- super(), 281, 282
- Swing, 186, 336, 381, 382, 425
 - JCheckBox, 442
 - JComponent, 426
 - JFrame, 426, 438
 - JList, 443
 - JPanel, 426
 - JTextArea, 440, 441
 - JTextField, 439
 - komponenty, 426
 - komponenty interakcyjne, 426
 - komponenty tła, 426
 - menedżery układu, 427
 - pasek przewijania, 440
 - zagnieżdżanie komponentów, 426
- switch, 694
- synchronizacja wątków, 534, 536, 546
 - wzajemna blokada wątków, 540
- synchronized, 534, 541
- syntezator, 368
- syntezator programowy, 345
- System, 184
- System.out.print(), 45
- System.out.println(), 41, 45, 684
- szkielet RMI, 636, 640

Ś

- ścisła kontrola typów, 51
- śmiertelny romb, 251
- średniki, 42, 44, 51

T

- tablice, 49, 91, 94, 115, 162, 165, 166, 186, 592, 596
 - długość, 186
 - elementy, 91
 - tablice tablic, 692
 - tablice wielowymiarowe, 692
 - tworzenie, 92
- tan(), 314
- TAR, 617
- TCP, 500, 509
- technologia CORBA, 636
- technologia Jini, 654
- technologia JWS, 621
- technologia RMI, 645
- test warunku, 43
- testowanie metod, 133
- testy logiczne, 43, 44
- this(), 284
- Thread, 513, 514, 516, 524
 - sleep(), 525, 526, 541
 - start(), 516
- throw, 351, 353
- throws, 351, 353
- Tiger, 36, 37
- toCharArray(), 691
- toLowerCase(), 691
- toString(), 237, 321, 560, 691
- toUpperCase(), 691
- transform(), 392
- transient, 463, 464, 466, 470
- TreeMap, 582
- TreeSet, 557, 558, 582, 588, 590
 - sortowanie, 590
- trim(), 691
- true, 43
- try-catch, 349, 352
 - sterowanie przepływem, 354
- try-finally, 355
- tworzenie
 - animacja, 408
 - archiwum JAR, 609
 - gniazda, 502

graficzny interfejs użytkownika, 380, 426
 grafika, 390
 interfejsy, 252, 253
 klasy, 131
 klasy abstrakcyjne, 228
 klasy wewnętrzne, 402
 klient, 510
 komparatory, 576
 metody abstrakcyjne, 231
 metody statyczne, 303
 namiastka RMI, 640
 obiekty, 68, 87, 212, 268, 270, 280
 obiekty klasy wewnętrznej, 404
 obiekty reprezentujące wartość, 315
 pakiety, 613
 połączenia sieciowe, 499
 serwer, 507
 serwlety, 648
 szkielet RMI, 640
 tablice, 92
 wątki, 513, 516, 527
 wykonywalne archiwum JAR, 609
 wykonywalne archiwum JAR zawierające pakiety, 616
 zdalne usługi, 637
 typy danych, 82, 110
 autoboxing, 317
 boolean, 83
 byte, 83
 char, 83
 double, 83
 float, 83
 int, 83
 liczby, 83
 long, 83
 łańcuchy znaków, 81
 mechanizm automatycznej konwersji, 317
 rzutowanie, 149
 short, 83
 wartości logiczne, 83
 znaki, 83
 typy MIME, 624

typy ogólne, 563, 564, 565, 592, 593
 argumenty, 565
 deklaracja zmiennych, 565
 kolekcje, 564
 metody, 568
 polimorfizm, 593, 594
 przypisanie zmiennych, 565
 rozszerzanie, 572
 zmienne, 565
 znaki wieloznaczne, 598
 typy podstawowe, 82, 83, 316
 automatyczna konwersja, 316
 klasy, 315
 niezmienność, 683
 typy sparametryzowane, 167
 typy wyliczeniowe, 693

V

uczenie się, 24
 układy graficznego interfejsu użytkownika, 396
 BorderLayout, 396
 ukrywanie danych, 113
 umieszczać klasy w pakiecie, 613
 umieszczać programów w archiwach JAR, 609
 UML, 29
 UnicastRemoteObject, 639, 645
 uruchamianie
 aplikacja za pośrednictwem przeglądarki WWW, 621
 archiwum JAR, 610
 program, 608, 614
 serwlet, 648
 testy, 135
 wątek, 515, 516, 527
 urządzenie MIDI, 368
 usługa lokalizacji Jini, 655
 usługi, 500, 629
 ustawianie
 dane, 111
 błędy, 158
 obiekty, 288
 odwołanie do obiektu, 288
 powtórzenia, 580

usypanie wątku, 514, 520, 525, 526
uzyskiwanie wartości na podstawie obiektu, 315

V

valueOf(), 691
varargs, 328
Vector, 582
void, 85, 106, 107

W

wartości domyślne składowych, 116
wartości liczbowe, 83
wartości logiczne, 43, 44, 83
wartości RGB, 276
wartości wyliczeniowe, 694
wartości wynikowe metody, 110, 111
wartość bezwzględna liczby, 314
wartość null, 94, 291
wartość statycznej zmiennej finalnej, 310
wątki, 513, 514, 524
 blokady, 535
 blokowanie, 520
 mechanizm zarządzający wątkami, 521
 operacje atomowe, 534
 pętla stanów, 520
 priorytety, 546
 problemy, 528
 run(), 516
 Runnable, 516, 517, 518
 sleep(), 520, 525, 541
 stan, 519
 start(), 516
 stos wywołań, 515
 stosowanie, 528
 synchonizacja, 534, 536, 538, 546
 synchronized, 534
 Thread, 513, 514
 tworzenie, 513, 516, 527
 tworzenie zadania, 518
 uruchamianie, 514, 515, 516, 527
 usypanie, 514, 525
 wątek główny, 514

 wątek realizacji, 519
 wątek uruchamialny, 519
 wątek zablokowany, 519
 współzawodnictwo, 528
 wzajemna blokada wątków, 540
 zarządzanie realizacją, 522
 wdrażanie aplikacji, 606
 aplikacje JWS, 624
 wdrażanie lokalne, 606
 wdrażanie zdalne, 606, 629
 wejście, 457
 while, 42, 43, 44, 147
 wiele odbiorców zdarzeń, 398
 wielokrotne dziedziczenie, 250, 257
 wielokrotne użycie kodu, 199, 210
 wielowątkowość, 513
 wielowierszowe pole tekstowe, 440
 wierzchołek stosu, 265
 windowClosing(), 388
 Windows, 382
 wirtualna maszyna Javy, 35, 42, 50, 264, 615
 write(), 471
 writeObject(), 456, 457, 461, 471
 współzawodnictwo wątków, 528
 WWW, 500
 wyjątki, 51, 343, 348, 352
 BindException, 501
 deklaracja, 351, 363, 365
 Exception, 350, 358
 finally, 355
 IOException, 471
 kolejność wielu bloków catch, 360
 MidiUnavailableException, 349, 366
 NullPointerException, 352
 NumberFormatException, 320
 obiekty, 350
 obsługa, 349, 365
 polimorficzny blok catch, 359
 polimorfizm, 358
 pomijanie wyjątków, 363
 przechwytywanie, 350
 przechwytywanie wielu wyjątków, 357

reguły stosowania, 366
RemoteException, 638, 639
 rezygnowanie z obsługi wyjątku, 364
RuntimeException, 352
 sterowanie przepływem w blokach try-catch, 354
throw, 351, 353
try-catch, 349, 352
try-finally, 355
 wiele bloków catch, 359
 wyjątki czasu wykonywania programu, 320
 wyjątki niesprawdzane, 352
 wyjątki sprawdzane, 352
 zgłaszanie, 351, 353
wyjście, 457
 wykonywalne archiwa JAR, 609, 610, 620
 wyliczenia, 693, 694, 695
 wymiary komponentu, 438
 wyrażenia logiczne, 181
 wyrażenie iteracyjne, 146
 wysyłanie danych, 498
wyświetlanie
 okno, 394
 plik JPEG, 389, 391
 ramka, 380, 426
 tekst, 45
wywołanie
 konstruktor, 269
 konstruktor klasy bazowej, 281
 metoda, 104, 108, 203, 245, 630
 metoda klasy bazowej, 208
 metoda obiektu na innym komputerze, 631
 metoda statyczna, 304
 metoda zdefiniowana w klasie bazowej, 256
 przeciążony konstruktor, 284
 super(), 284
 this(), 284
 wywołanie zdalnych metod, 467, 632, 635
 wzajemna blokada wątków, 540

X

XOR, 682
XP, 133

Y

YEAR, 334

Z

zaginione odwołania, 99
 zagnieżdżanie komponentów, 426
 zakończenie wykonywania pętli, 137, 141
 zamiana liczby nałańcuch znaków, 321
 zamykanie strumienia, 456
 zaokrąglanie liczb, 314
zapisywania
 dane w gnieździe, 503
 kompozycja, 487
 łańcuch znaków w pliku tekstowym, 471
 obiekty, 454, 456
 pliki, 471, 476
 pliki tekstowe, 455, 472
 stan obiektu, 455
 zapobieganie konfliktom nazw pakietów, 612
 zarządzanie realizacją wątków, 522
 zarządzanie wątkami, 521
zasięg, 286
 zasięg blokowy, 685
 zmienne globalne, 73
 zmienne lokalne, 117, 287
Zatopić portal, 128
zbiory, 581
 TreeSet, 588
 zdalne usługi, 637
 zdalne wdrażanie, 629
zdarzenia, 382, 387, 394, 414
 ActionEvent, 385, 386
 ActionListener, 385
 ControllerEvent, 416
 KeyEvent, 388
 klasy, 384
 metody obsługi zdarzeń, 388
 MouseEvent, 384
 MouseListener, 384
 mysz, 384
 obiekt zdarzenia, 384
 obsługa, 383, 388

zdarzenia

odbieranie zdarzeń, 416
odbiorca, 384, 385, 387
przechwytywanie, 386
rejestracja odbiorcy, 416
system nazewnictwa, 388
wiele odbiorców zdarzeń, 398
źródło zdarzeń, 385, 387
zdarzenia MIDI, 368, 371, 375
komunikaty, 372
zagłaszczenie wyjątków, 348, 351, 353
ZIP, 617
zmiana specyfikacji, 61
zmiany w kodzie, 62
zmienna lista argumentów, 328
zmienne, 42, 44, 81, 87, 94
 deklaracja, 71, 82
 nazwy, 82, 85, 310
 porównywanie, 118
 przypisanie wartości, 84
 typy danych, 82

wartość, 86
zmienne finalne, 311
zmienne globalne, 73
zmienne logiczne, 44
zmienne składowe, 66
zmienne typów ogólnych, 565
zmienne lokalne, 117, 264, 286
 cykl istnienia, 286, 292
 czas istnienia, 286, 287
 zasięg, 287
zmienne referencyjne, 86, 94, 212, 268
 cykl istnienia, 288
 Object, 257
znacznik czasu, 331
znaki, 83, 327
znaki wieloznaczne, 335, 598
ZONE_OFFSET, 334
zwracanie wartości z metod, 107

Ź

źródło zdarzeń, 384, 385, 387, 388

