helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Head First Java. Edycja polska

Autorzy: Kathy Sierra, Bert Bates
Tłumaczenie: Piotr Rajca
ISBN: 83-7361-413-3
Tytuł oryginału: Head First Java
Format: B5, stron: 648
Przykłady na ftp: 102 kB

Otwórz swój umysł. Poznaj język Java w niekonwencjonalny sposób. Nie będziesz musiał przebijać się przez długie wywody i czytać kilometrowych przykładów. Autorzy książki „Head First Java. Edycja polska" wybrali inny sposób przedstawienia czytelnikom najpopularniejszego języka programowania ery internetu – uniwersalnego, obiektowego, o prostej i czytelnej składni, a co najważniejsze – dostępnego nieodpłatnie na wszystkich platformach systemowych i sprzętowych.

W „Head First Java. Edycja polska" znajdziesz układanki, ilustrowane wyjaśnienia, zagadki oraz wywiady z najważniejszymi obiektami Javy. W ten nietypowy sposób nauczysz się zasad programowania zorientowanego obiektowo w języku Java. Z książki dowiesz się niemal wszystkiego – zaczynając od zagadnień zupełnie podstawowych, a kończąc na bardzo zaawansowanych, takich jak wątki, połączenia sieciowe oraz programowanie rozproszone przy wykorzystaniu technologii RMI. Najważniejsze jest jednak to, że nauczysz się myśleć jak programista używający obiektowych języków programowania.

- Podstawowe informacje o języku Java
- Klasy i obiekty
- Typy danych
- Pierwszy program w Javie
- Java API
- Programowanie obiektowe – dziedziczenie, polimorfizm, interfejsy i klasy abstrakcyjne
- Metody
- Obsługa wyjątków
- Graficzny interfejs użytkownika
- Operacje wejścia-wyjścia
- Programowanie sieciowe i RMI

Jeśli chcesz poznać Javę i świetnie się przy tym bawić – kup tę książkę.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści (skrócony)

	Wprowadzenie	17
1.	Przełamując zalew początkowych trudności — szybki skok na głęboką wodę	27
2.	Wycieczka do Obiektowa — tak, tam będą same obiekty	51
3.	Poznaj swoje zmienne — typy podstawowe i referencje	73
4.	Jak działają obiekty — stan obiektu wpływa na działanie metod	95
5.	Supermocne metody — sterowanie przepływem, operacje i nie tylko	119
6.	Korzystanie z biblioteki Javy — nie musisz pisać tego sam	147
7.	Wygodniejsze życie w Obiektowie — planowanie swojej przyszłości	185
8.	Poważny polimorfizm — wykorzystanie abstrakcyjnych klas i interfejsów	217
9.	Życie i śmierć obiektu — konstruktory i zarządzanie pamięcią	255
10.	Liczby mają znaczenie — klasa Math, formatowanie, klasy „opakowujące” i składowe statyczne	293
11.	Ryzykowne działania — obsługa wyjątków	321
12.	Historia bardzo graficzna — wprowadzenie do GUI, obsługi zdarzeń i klas wewnętrznych	357
13.	Popracuj nad Swingiem — menedżery rozkładu i komponenty	403
14.	Zapisywanie obiektów — serializacja i operacje wejścia-wyjścia	431
15.	Nawiąż połączenie — połączenia sieciowe i wielowątkowość	471
16.	Rozpowszechnij swój kod — pakowanie i publikowanie	529
17.	Przetwarzanie rozproszone — RMI z odrobinką serwletów, EJB i Jini	553
A	Ostatnie doprawianie kodu	595
B	Dziesięć najważniejszych zagadnień, które niemal znalazły się w tej książce...	605
	Skorowidz	625

Spis treści (na serio)

10 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 długo. Myśli sobie: „Lepiej zostawić miejsce na ważniejsze rzeczy, takie jak dzikie zwierzęta, których należy się wystrzegać, lub rozważania, czy jeżdżenie 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?	18
Co sobie myśli Twój mózg?	19
Metapoznanie	21
Zmuś swój mózg do posłuszeństwa	23
Czego potrzebujesz, aby skorzystać z tej książki?	24

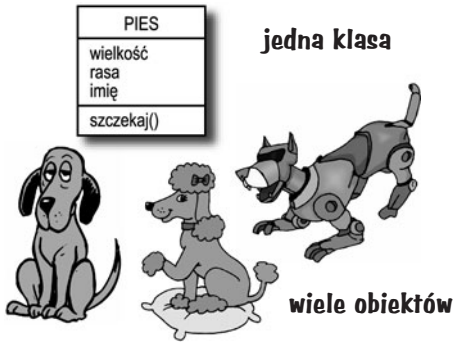
1 Przełamując zalew początkowych trudności

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? 28
Struktura kodu w Javie 31
Anatomia klasy 32
Metoda **`main()`** 33
Pętle 35
Rozgałęzienia warunkowe (test *if*) 37
Tworzenie aplikacji „99 butelek piwa" 38
Program krasomówczy 41
Pogawędki przy kominku — spór kompilatora z JVM 42
Ćwiczenia i zagadki 44

2 Wycieczka do Obiektowa

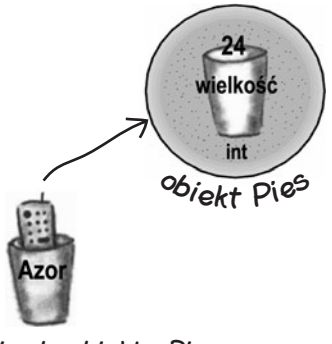
Mówiono mi, że będą tam same obiekty. W rozdziale 1. cały kod programu został umieszczony w metodzie **`main()`**. Nie jest to w pełni zgodne z zasadami programowania obiektowego. A zatem trzeba porzucić świat programowania proceduralnego i rozpocząć tworzenie własnych obiektów. Zobaczymy, co sprawia, 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 (Jurek „obiektowiec" kontra Bronek „proceduralny") 52
Dziedziczenie (wprowadzenie) 55
Przesłanianie metod (wprowadzenie) 56
Czym jest klasa? (metody, składowe) 58
Tworzenie pierwszego obiektu 60
Użycie metody **`main()`** 62
Kod od kuchni 63
Ćwiczenia i zagadki 66

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.



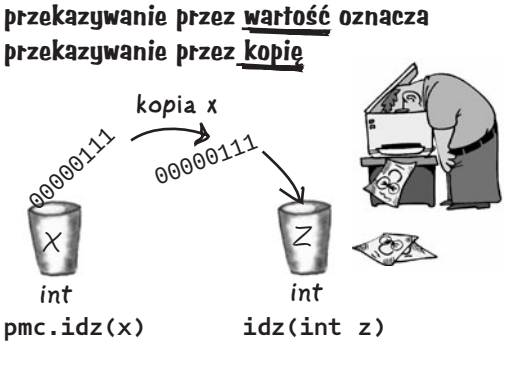
odwołanie do obiektu Pies

Deklarowanie zmiennej (Java zwraca uwagę na *typ*)	74
Typy podstawowe („Poproszę podwójne z dodatkową pianką”)	75
Słowa kluczowe Javy	77
Odwołania (zdalne sterowanie obiektami)	78
Deklarowanie i przypisywanie obiektów	79
Obiekty na automatycznie sprzątanej stercie	81
Tablice (pierwszy rzut oka)	83
Ćwiczenia i zagadki	87

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 kilo, szczekamy radośnie, w przeciwnym przypadku...”. ***Spróbujmy zmienić stan obiektu!***

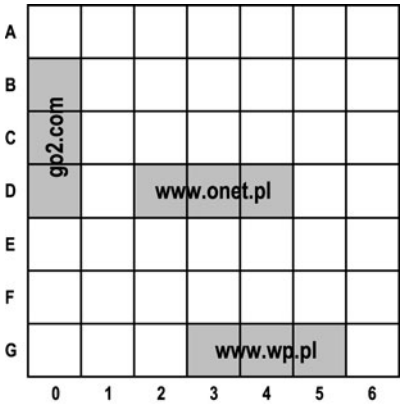
Metody wykorzystują stan obiektu (obiektowy pies szczeka inaczej)	97
Argumenty metod i typy zwracanych wartości	98
Przekazywanie przez wartość (zmienna zawsze jest kopiowana)	101
Metody pobierające i zapisujące	103
Hermetyzacja (użyj jej lub zaryzykuj upokorzenie)	104
Wykorzystanie odwołań w tablicy	107
Ćwiczenia i zagadki	112



5 Supermocne metody

Dodajmy naszym metodom nieco siły. Zajmowaliśmy się już zmiennymi, eksperymentowali z kilkoma obiektami i napisali kod paru programów. Jednak potrzeba nam więcej narzędzi. Na przykład **operatorów**. I pętli. Może przydałoby się **wygenerować kilka liczb losowych** i **zamienić łańcuch znaków na liczbę całkowitą**. O tak, to byłoby super. I dlaczego nie zajmiemy się stworzeniem prawdziwego programu, aby zobaczyć (i doświadczyć), jak wygląda kodowanie w Javie? **Może jakąś grę**, taką jak „Zatopić portal" (podobną do gry w okręty).

Stworzymy grę Zatopić portal



Tworzenie gry Zatopić portal 120
Rozpoczynanie pracy nad prostą wersją gry Zatopić portal 122
Pisanie kodu przygotowawczego (pseudokodu gry) 123
Kod testowy prostej wersji gry 126
Pisanie kodu prostej wersji gry 127
Ostateczny kod prostej wersji gry Zatopić portal 130
Generowanie liczb losowych przy użyciu `Math.random()` 135
Gotowy kod do pobierania danych wpisywanych w wierszu poleceń 136
Realizacja cykliczna przy użyciu pętli *for* 138
Rzutowanie dużych typów prostych na mniejsze 140
Konwersje łańcuchów znaków na liczby przy użyciu `Integer.parseInt()` 140
Ćwiczenia i zagadki 141

6 Korzystanie z biblioteki Javy

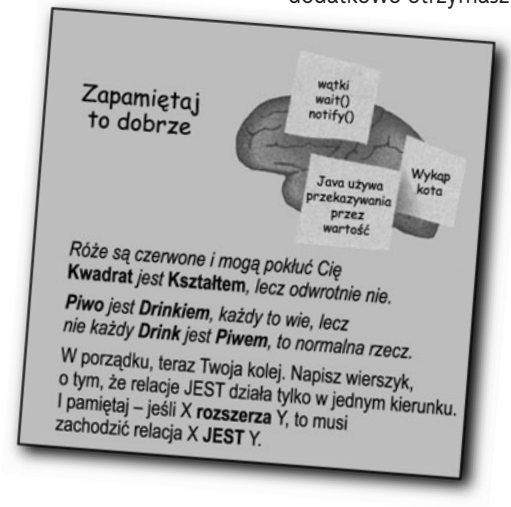
Java jest wyposażona w setki gotowych klas. Jeśli tylko dowiesz się, jak znaleźć w bibliotece Javy (nazywanej także **Java API**) to, czego szukasz, nie będziesz musiał ponownie wymyślać koła. *W końcu masz ciekawsze rzeczy do roboty.* Jeśli masz zamiar napisać program, to równie dobrze możesz napisać 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.

*„Jak to dobrze, że w pakiecie java.util dostępna jest klasa ArrayList. Ale jak mógłbym się o tym **sam** dowiedzieć?"*
— Julian, 31 lat, model

Analizowanie błędu w prostej wersji gry Zatopić portal 148
ArrayList (wykorzystanie Java API) 154
Poprawa kodu klasy Portal 160
Tworzenie *właściwej* wersji gry (Zatopić portal) 162
Pseudokod *właściwej* wersji gry 166
Kod właściwej wersji gry 168
Wyrażenia *logiczne* (*boolowskie*) 173
Korzystanie z biblioteki (Java API) 176
Korzystanie z pakietów (instrukcja importu i pełne nazwy) 177
Korzystanie z dokumentacji w wersji HTML i literatury 180
Ćwiczenia i zagadki 182

7 Wygodniejsze życie w Obiektowie

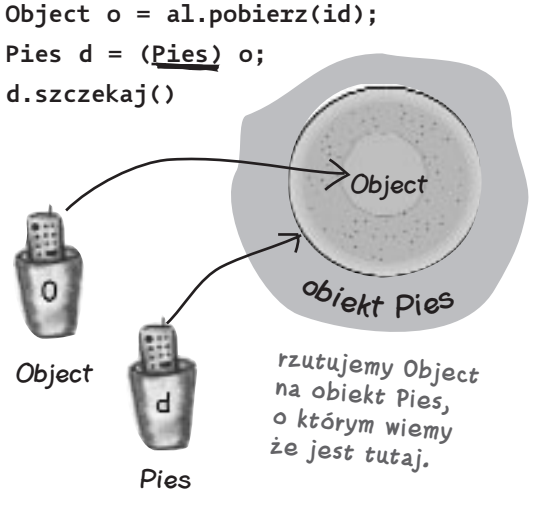
Planuj swoje programy, myśląc o przyszłości. A gdyby tak można tworzyć kod, który inni programiści mogliby rozbudowywać, i to w **prosty sposób**? A gdyby tak można było tworzyć elastyczny kod z myślą o tych najnowszych zmianach wprowadzanych w specyfikacji Javy? Kiedy dowiesz się o planie polimorficznym, 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 wykorzystywania dziedziczenia.



Zrozumienie dziedziczenia (relacje między klasami nadrzędnymi i podrzędnymi) 188
Projektowanie drzewa dziedziczenia (symulacja na zwierzętach) 190
Unikanie powielania kodu (przy wykorzystaniu dziedziczenia) 191
Przesłanianie metod 192
JEST i MA (dziewczyna z wanną) 197
Co dziedziczysz po klasie nadrzędnej? 200
Co w rzeczywistości daje nam dziedziczenie? 202
Polimorfizm (używanie odwołania klasy nadrzędnej do obiektu klasy podrzędnej) 203
Reguły przesłaniania (nie dotykaj tych argumentów i typu wartości wynikowej) 210
Przeciążanie metody (powtórnie użyta nazwa metody, nic więcej) 211
Ćwiczenia i zagadki 213

8 Poważny polimorfizm

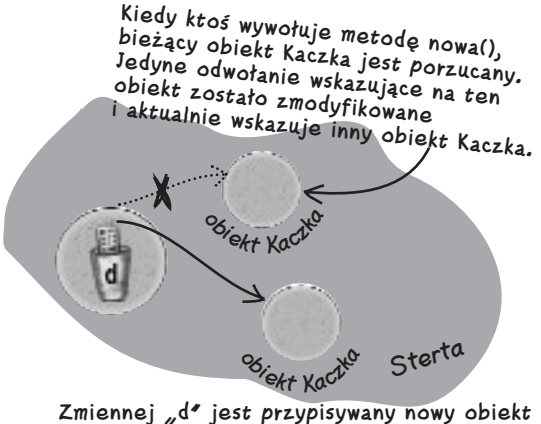
Dziedziczenie to tylko początek. Aby w pełni wykorzystać polimorfizm, niezbędne nam będą interfejsy. Musimy pójść dalej, poza proste dziedziczenie, i uzyskać elastyczność, jaką może zapewnić wyłącznie projektowanie i kodowanie z wykorzystaniem interfejsów. Czym jest interfejs? W 100% abstrakcyjną klasą.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



Obiektów niektórych klas po prostu nie należy tworzyć 220
Klasy abstrakcyjne (*nie* dają możliwości tworzenia obiektów) 221
Metody abstrakcyjne (muszą zostać zaimplementowane) 223
Polimorfizm w działaniu 226
Klasa Object (ostateczna klasa nadrzędna) 228
Odczytywanie obiektów z listy `ArrayList` (są odczytywane jako `Object`) 231
Kompilator sprawdza typy odwołań (nim pozwoli na wywołanie metody) 233
Połącz się ze swoim wewnętrznym Obiect-em 234
Odwołania polimorficzne 235
Rzutowanie odwołania do obiektu (przesuwanie się w dół drzewa dziedziczenia) 236
„Śmiertelny romb" (problem wielokrotnego dziedziczenia) 243
Wykorzystanie interfejsów (najlepsze rozwiązanie!) 244
Ćwiczenia i zagadki 249

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*. **Odśmiecacz 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 nadrzędnych, odwołaniach pustych oraz przydatności odśmiecacza pamięci.



Stos i sterta, gdzie mieszkają zmienne i obiekty	256
Metody na stosie	257
Gdzie są przechowywane zmienne *lokalne*?	258
Gdzie są przechowywane *składowe*?	259
Cud uworzenia obiektu	260
Konstruktory (kod wykonywany, gdy powiemy: „Stań się")	261
Inicjalizacja stanu nowego obiektu Kaczka	263
Kompilator potrafi stworzyć konstruktor domyślny (bezargumentowy)	265
Konstruktory przeciążone	267
Konstruktory klas nadrzędnych (łańcuch wywołań konstruktorów)	270
Wywoływanie przeciążonych konstruktorów przy użyciu *this()*	276
Istnienie obiektu	278
Odśmiecanie (i zapewnianie przydatności obiektu)	280
Ćwiczenia i zagadki	287

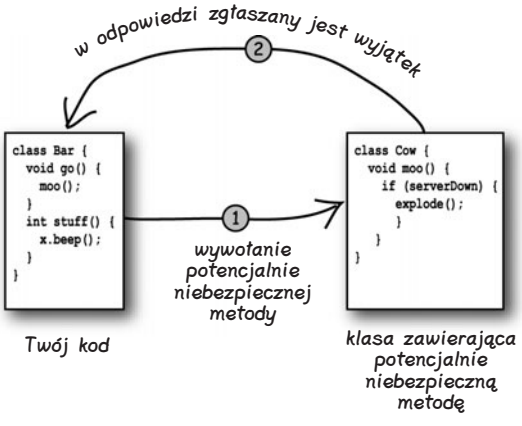
10 Liczby mają znaczenie

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*, omówimy także stałe w Javie — statyczne zmienne *sfinalizowane*.

Klasa `Math` (czy naprawdę potrzeba nam obiektu tej klasy?)	294
Metody statyczne	295
Zmienne statyczne	297
Stałe (statyczne zmienne sfinalizowane)	302
Metody klasy `Math` (`random()`, `round()`, `abs()` i inne)	306
Klasy „opakowujące" (`Integer`, `Boolean`, `Character` itd.)	307
Formatowanie liczb	310
Formatowanie dat	312
Ćwiczenia i zagadki	315

11 Ryzykowne działania

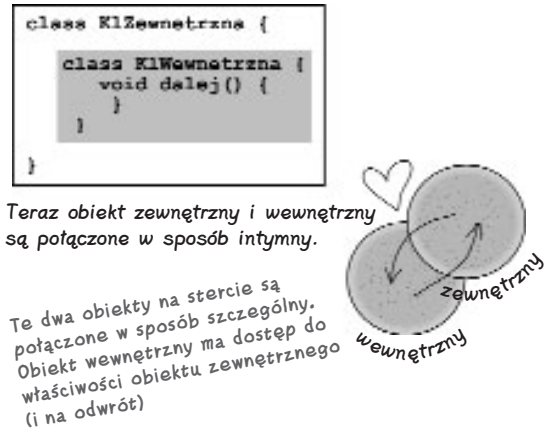
Czasami zdarzają się nieprzewidziane sytuacje. Pliku nie ma tam, gdzie powinien być. Serwer został wyłączony. Niezależnie od tego, jak dobrym jesteś programistą, nie jesteś w stanie kontrolować *wszystkiego*. Kiedy tworzysz metodę, której działanie jest opatrzone ryzykiem niepowodzenia, musisz także stworzyć kod, który obsłuży potencjalnie niebezpieczną sytuację. Ale skąd wiadomo, że metoda może nieść ze sobą potencjalne niebezpieczeństwo? Gdzie umieszczać kod *obsługujący* **wyjątkowe** sytuacje? W tym rozdziale stworzymy odtwarzacz MIDI wykorzystujący ryzykowny interfejs programistyczny `JavaSound`, zatem lepiej dowiedzmy się, jak zabezpieczyć się przed potencjalnymi niebezpieczeństwami.



Tworzenie maszyny muzycznej (MuzMachina) 322
A co, jeśli trzeba wywołać potencjalnie niebezpieczny kod? 325
Wyjątki informują, że „mogło się stać coś złego” 326
Kompilator gwarantuje (sprawdza), że będziesz świadom zagrożeń 327
Przechwytywanie wyjątków przy użyciu `try-catch` (deskorolkarz) 328
Sterowanie przepływem w blokach `try-catch` 332
Blok `finally` (niezależnie od tego, co się dzieje, wyłącz piekarnik!) 333
Przechwytywanie wielu wyjątków (kolejność ma znaczenie) 335
Deklarowanie wyjątków (pomiń go) 341
Prawo obsługuj lub deklaruj 343
Kod od kuchni (generowanie dźwięków) 345
Ćwiczenia i zagadki 354

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.



Twój pierwszy interfejs graficzny 359
Przechwytywanie zdarzeń generowanych przez użytkownika 361
Implementacja interfejsu odbiorcy 362
Przechwytywanie zdarzenia `ActionEvent` przycisku 364
Tworzenie interfejsu użytkownika wykorzystującego grafikę 367
Zabawa z metodą `paintComponent()` 369
Obiekt `Graphics2D` 370
Umieszczanie na ekranie więcej niż jednego przycisku 376
Pomoc ze strony klas wewnętrznych (stwórz odbiorcę jako klasę wewnętrzną) 382
Animacja (przesuń to, narysuj to, przesuń to, narysuj to i tak dalej) 386
Kod od kuchni (rysowanie przy dźwiękach muzyki) 390
Ćwiczenia i zagadki 398

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 popatrzysz na wyniki, to często będziesz mógł sobie pomyśleć: „Hej, przecież to nie miało być wyświetlone 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 menedżer 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 wielkość.

Także komponenty na północy i południu mają preferowaną wielkość.

Komponenty biblioteki Swing 404
Menedżery układu (kontrolują wielkość i rozmieszczenie) 405
Trzy menedżery układu (`BorderLayout`, `FlowLayout`, `BoxLayout`) 407
`BorderLayout` (zarządza pięcioma regionami) 408
`FlowLayout` (zwraca uwagę na kolejność i preferowaną wielkość) 412
`BoxLayout` (podobny do poprzedniego, lecz może rozmieszczać elementy w pionie) 415
`JTextField` (służy do wpisywania pojedynczego wiersza tekstu) 417
`JTextArea` (służy do wpisywania wielu wierszy tekstu i ma możliwość jego przewijania) 418
`JCheckBox` (czy ta opcja jest wybrana?) 420
`JList` (lista elementów z możliwością przewijania) 421
Kod od kuchni (coś wielkiego — tworzenie klienta pogawędek aplikacji MuzMachina) 422
Ćwiczenia i zagadki 428

14 Zapisywanie obiektów

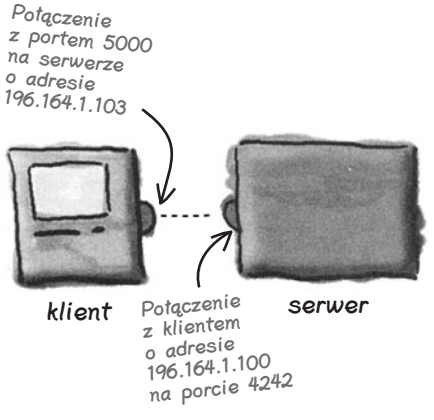
Obiekty można pakować i odtwarzać. Obiekty mają swój stan i działanie. Działanie obiektu określa jego *klasa*, natomiast stan zależy od konkretnego obiektu. Jeśli program musi zapisać stan *obiektu*, możesz to zrobić w sposób *trudny* — analizując każdy obiekt i pracowicie zapisując wartości wszystkich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

jakieś pytania?

Stan obiektów biblioteki Swing 433
Zapisywanie zserializowanego obiektu do pliku 434
Strumienie wejścia-wyjścia w Javie 437
Serializacja obiektów 438
Implementacja intefejsu `Serializable` 439
Stosowanie zmiennych pomijanych podczas serialiacji 441
Deserializacja obiektów 443
Zapis zawartości pliku tekstowego 447
java.io.File 452
Odczyt zawartości pliku tekstowego 454
StringTokenizer 458
Kod od kuchni 462
Ćwiczenia i zagadki 466

15 Nawiąż połączenie

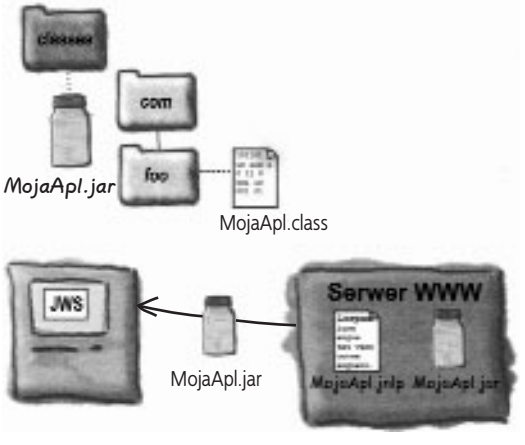
Nawiąż połączenie ze światem zewnętrznym. To takie łatwe. Wszelkie szczegóły związane z komunikacją sieciową na niskim poziomie są obsługiwane przez klasy należące do biblioteki java.net. Jedną z najlepszych cech Javy jest to, iż wysyłanie i odbieranie danych przez sieć to w zasadzie normalne operacje 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*?



Ogólne omówienie programu klienta	473
Nawiązywanie połączenia, wysyłanie i odbieranie	474
Gniazda sieciowe	475
Porty TCP	476
Odczytywanie danych z gniazda (przy użyciu **`BufferedReader`**)	478
Zapisywanie danych w gnieździe (przy użyciu **`PrintWriter`**)	479
Pisanie programu **`CodziennePoradyKlient`**	480
Tworzenie prostego serwera	483
Kod aplikacji **`CodziennePoradySerwer`**	484
Tworzenie klienta pogawędek	486
Wiele stosów wywołań	490
Uruchamianie nowego wątku (utworzenie i rozpoczęcie działania)	492
Interfejs **`Runnable`** (zadanie, jakie wątek ma wykonać)	493
Trzy stany nowego obiektu **`Thread`** (nowy, uruchamialny, działający)	495
Pętla działania	496
Mechanizm szeregowania wątków (to on decyduje, a nie Ty)	497
Usypianie wątku	501
Tworzenie i uruchamianie dwóch wątków	503
Zagadnienia współbieżności — czy te problemy można rozwiązać?	505
Problem współbieżności Romka i Moniki	506
Blokowanie w celu stworzenia elementów atomowych	510
Każdy obiekt ma blokadę	511
Przerażający problem „utraconej modyfikacji”	512
Metody synchronizowane (przy użyciu blokowania)	514
Wzajemna blokada!	516
Kod wielowątkowego klienta pogawędek	518
Gotowy do użycia **`ProstyKlientPogawedek`**	520
Ćwiczenia i zagadki	524

16 Rozpowszechnij swój kod

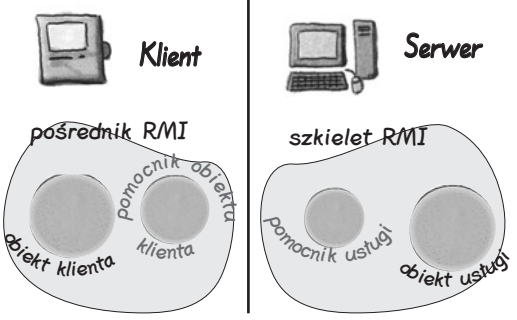
Już czas wypłynąć na „szerokie wody". Napisałeś kod swojej aplikacji. Przetestowałeś go. Udoskonaliłeś. Powiedziałeś wszystkim znajomym, że świetnie by było, gdybyś już w życiu nie musiał napisać choćby jednej linijki kodu. Ale w końcu stworzyłeś dzieło sztuki. Przecież Twoja aplikacja działa! W ostatnich 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 poszczycić Java, są prostsze, niż można by przypuszczać.



Możliwości wdrażania 530
Niezależne przechowywanie kodu źródłowego i plików klasowych 532
Tworzenie wykonywalnych plików *JAR* (archiwów Javy) 533
Uruchamianie wykonywalnych plików *JAR* 534
Umieszczaj swoje klasy w pakietach! 535
Zapobieganie konfliktom nazw pakietów 536
Pakietom musi odpowiadać odpowiednia struktura katalogów 537
Kompilacja i uruchamianie programu, w którym wykorzystywane są pakiety 538
Kompilacja z opcją **-d** 539
Tworzenie wykonywalnych plików *JAR* (z wykorzystaniem pakietów) 540
Wdrażanie przez internet przy użyciu technologii *JWS* (ang. *Java Web Start*) 545
Plik *.jnlp* 547
Jak tworzyć i wdrażać aplikacje JWS? 548
Ćwiczenia i zagadki 549

17 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.

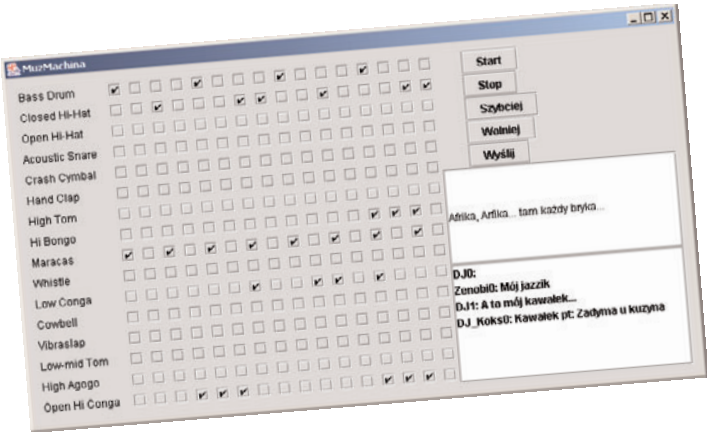


Wywoływanie zdalnych metod (RMI), bardzo szczegółowy opis praktyczny 554-570
Serwlety (*pobieżna prezentacja*) 571
Enterprise JavaBeans (komponenty EJB, bardzo pobieżna prezentacja) 577
Jini, najlepsza z możliwych sztuczek 578
Tworzenie naprawdę fajnej, uniwersalnej przeglądarki usług 582
Koniec 594

Dodatek A

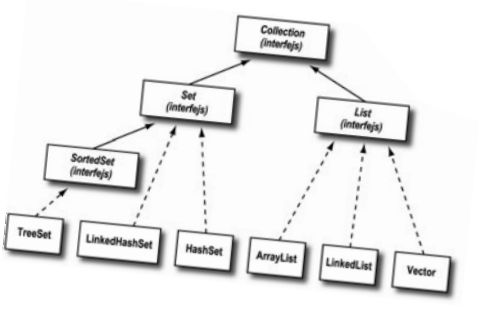
Ostatnie doprawianie kodu. Kompletny kod aplikacji MuzMachina w ostatecznej wersji klient-serwer z możliwością prowadzenia muzycznych pogawędek.

MuzMachinaKońcowa (kod klienta) 596
SerwerMuzyczny (kod serwera) 603



Dodatek B

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



Operacje bitowe 606
Niezmienność 607
Asercje 608
Zasięg blokowy 609
Połączone wywołania 610
Przesłanianie metody equals() 611
Poziomy dostępu oraz modyfikatory dostępu (czyli co kto widzi) 612
Metody klas String oraz StringBuffer 614
Tablice wielowymiarowe 615
Kolekcje 616

Skorowidz

625

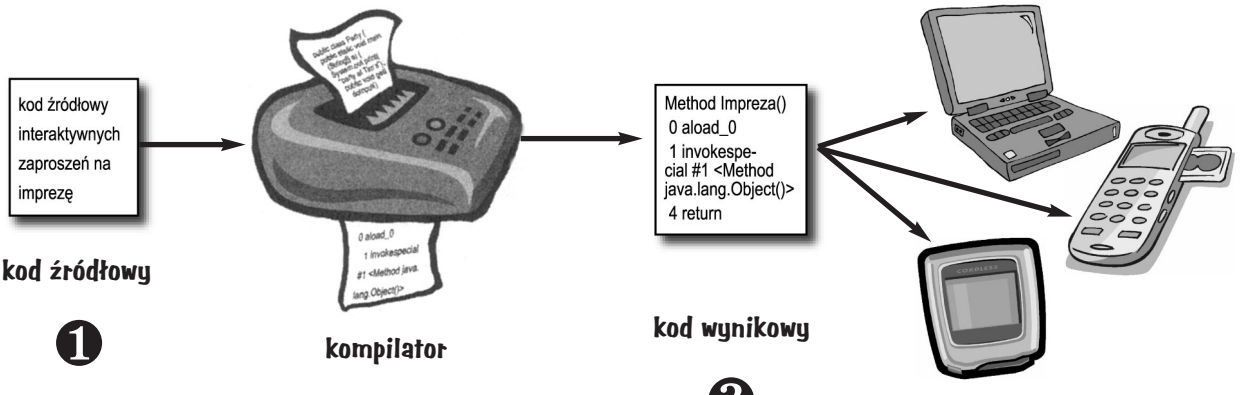
Przełamując zalew początkowych trudności

Wskakujcie, woda jest świetna! Napiszemy nieco kodu, skompilujemy go i uruchomimy. Porozmawiamy także trochę o składni, o pętlach, rozgałęzieniach oraz innych sprawach, które sprawiają, że Java jest taka fajna. Zanim się obejrzycie już będziecie pisać programy.

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 — uruchamiaj wszędzie"**, jest po prostu zbyt duża. Nagle pojawiło się wielu oddanych użytkowników Javy, choć programiści walczyli z błędami, ograniczeniami i, 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 aplet. Ale Ty, Ty masz do dyspozycji **udoskonaloną**, **szybszą** i o niebo potężniejszą, nowoczesną wersję języka. I pamiętaj — pionierzy Javy nie mieli do dyspozycji tej książki. Cóż, było im trudniej.

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.



kod źródłowy

1

Stworzenie dokumentu źródłowego. Wykorzystywany jest przy tym określony protokół (w tym przypadku — język Java).

kompilator

2

Przetworzenie dokumentu przy użyciu kompilatora kodu źródłowego. Kompilator sprawdza, czy w programie nie ma błędów, i nie dopuści do jego skompilowania aż do momentu, gdy uzyska pewność, że program będzie działać dobrze.

kod wynikowy

3

Kompilator tworzy nowy dokument, którego zawartość stanowi skompilowany ***kod bajtowy*** Javy. Java będzie w stanie zinterpretować (lub przetłumaczyć) ten dokument do postaci, którą będzie w stanie wykonać. Skompilowany kod bajtowy są niezależne od platformy systemowej.

maszyny wirtualne

4

Twoi przyjaciele nie mają fizycznej maszyny Javy, dysponują jednak maszynami ***wirtualnymi*** (zaimplementowanymi w formie programów) działającymi w różnego typu gadżetach elektronicznych. Te wirtualne maszyny odczytują i wykonują kody bajtowe.

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.

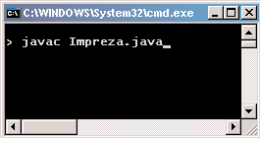
```
import java.awt.*;
import java.awt.event.*;
class Impreza {
  public void twórzZaproszenie() {
    Frame f = new Frame();
    Label l = new Label("Impreza u Tomka");
    Button b = new Button("Pewnie!");
    Button c = new Button("eeee ...")
    Panel p = new Panel();
    p.add(l);
}  // więcej kodu...
```

kod źródłowy

1

Wpisz kod źródłowy.

Zapisz go jako ***Impreza.java***.



kompilator

2

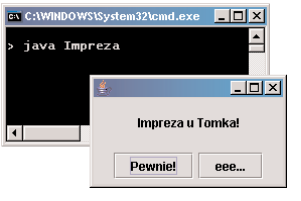
Skompiluj plik ***Impreza.java***, uruchamiając program `javac` (czyli kompilator). Jeśli nie pojawią się żadne błędy, wygenerowany zostanie drugi plik o nazwie ***Impreza.class***. Wygenerowany przez kompilator plik Impreza.class zawiera kody bajtowe.

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

kod wynikowy

3

Skompilowany kod: ***Impreza.class***.

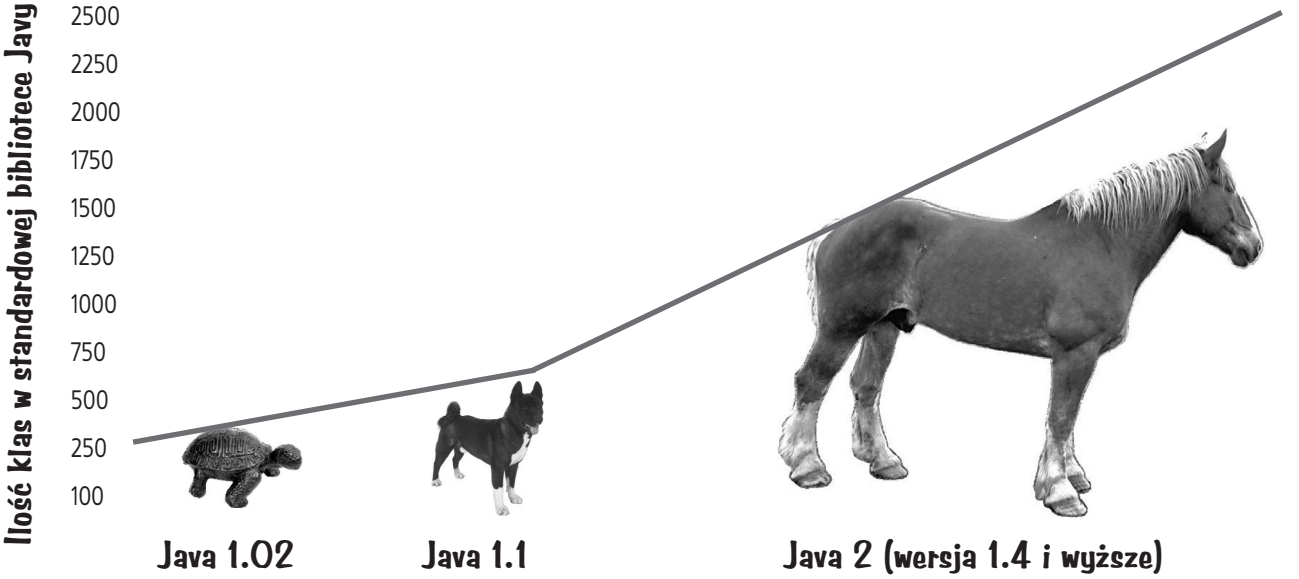


maszyny wirtualne

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.

Krótka historia Javy



Java 1.02

250 klas

Wolna

Fajna nazwa i logo. Jej używanie dawało dużo radości. Masa błędów. Zdecydowanie najciekawsze są ***aplety***.

Java 1.1

500 klas

Nieco szybsza

Więcej możliwości, nieco ***bardziej przyjazna*** dla programistów. Staje się ***bardzo popularna***. Lepszy kod do obsługi graficznego interfejsu użytkownika (GUI).

Java 2 (wersja 1.4 i wyższe)

2300 klas (Java 1.4)

Dużo szybsza; może (czasami) działać z szybkością dorównująca szybkości działania programów kompilowanych 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 i internetowych.

Zaostrz ołówek

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 < 25) mojPies.szczekaj(8);
while (x > 3) {
  mojPies.zabawa();
}
```

Czy potrafisz określić, do czego służą poszczególne wiersze kodu?

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

Hej, ruszcie głowami!

Struktura kodu w Javie

Umieść definicję klasy w pliku źródłowym.

W klasie umieść definicje metod.

W metodach umieść instrukcje.

Co się umieszczca 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 klasę).

```
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;

   }
}
```

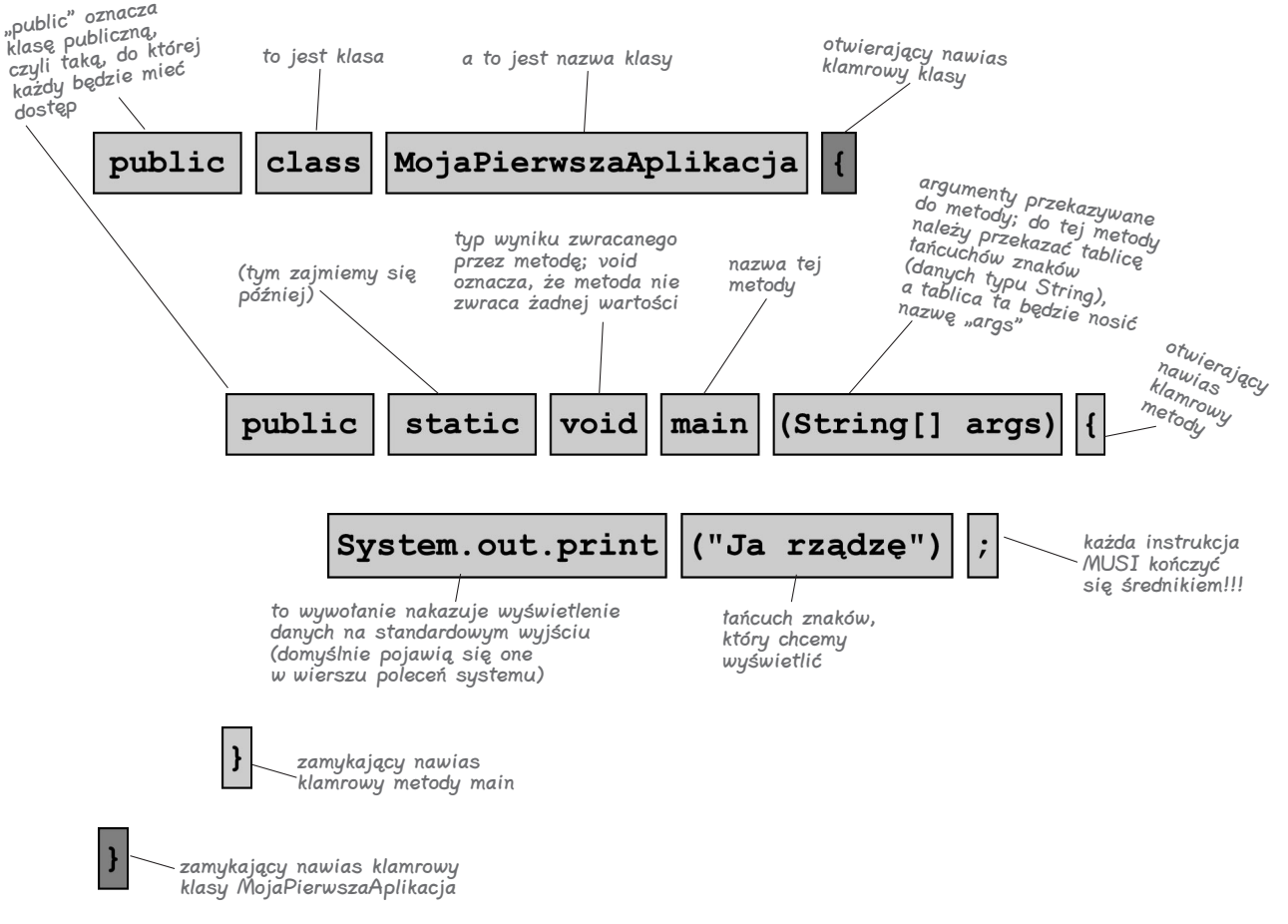
instrukcje

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*).



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 **`Witamy`**, a następnie zacznij wykonywać metodę **`main()`**. Program ma działać aż do momentu wykonania całego kodu metody **`main()`**”.

```
public class MojaPierwszaAplikacja {

   public static void main
(String[] args) {

   System.out.print("Ja rządzę");

 }

}
```

W rozdziale 2. dokładniej zajmiemy się wszystkimi zagadnieniami związanymi z *klasami*, jednak 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()`**.

MojaPierwszaAplikacja.java

kompilator

```
Method MojaPierwszaAplikacja()  0 aload_0
1
invokespecial #1 <Method
java.lang.Object()>

  4 return

Method void
main(java.lang.String[])

  0 getstatic #2 <Field
```

MojaPierwszaAplikacja.class

```
public class MojaPierwszaAplikacja {

  public static void main (String[] args) {
    System.out.println("Ja rządzę!");
    System.out.println("Światem!");

  }

}
```

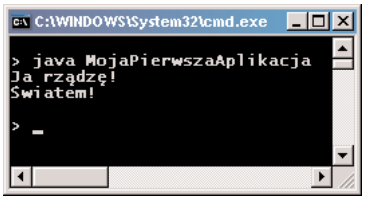
1 Zapisz

`MojaPierwszaAplikacja.java`

2 Skompiluj

`javac MojaPierwszaAplikacja.java`

3 Wykonaj



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
```

2 Wykonała coś wielokrotnie

Pętle: *for* oraz *while*

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

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

3 Wykonała coś pod pewnym warunkiem

Rozgałęzienia: testy *if* oraz *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!");
```

Składnia na wesoło

★ Każda instrukcja musi się kończyć średnikiem.

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

★ Komentarze jednowierszowe zaczynają się do dwóch znaków ukośnika.

```
// 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 4.).

```
int waga;
// typ: int (liczba całkowita),
// nazwa: waga
```

★ Definicje klas i metod należy umieszczać wewnątrz nawiasów klamrowych.

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

```
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 usnąłeś z nudów. Wszystko, co jest umieszczone wewnątrz *bloku* pętli, jest wykonywane dopóty, dopóki pewien warunek logiczny jest spełniony. Blok pętli jest wyznaczany przez parę nawiasów klamrowych, a zatem wewnątrz 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*) *wKubeczkuSąLody* 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 (zapisywanym jako *dwa* znaki równości). Wielu programistów niechcąco wpisuje **=** 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; // pętla będzie wykonywana
               w nieskończoność
}
int z = 27;
while (z == 17) {
  // kod pętli nie zostanie wykonany, gdyż
  // z nie jest równe 17
}
```

Nie ma niemądrych pytań

P: Dlaczego wszystko musi być umieszczane wewnątrz klasy?

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...");
  }
}
```

A oto wyniki

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

KLUCZOWE ZAGADNIENIA

- 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 browar…” stwierdzamy: „***jeśli*** ciągle mamy browar…”.

```
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");
  }
}
```

Wyniki wykonania programu

```
% java TestIf
x musi mieć wartość 3
Ta instrukcja zawsze zostanie wykonana
```

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");
  }
}
```

Wyniki wykonania nowego programu

```
% java TestIf2
x jest RÓŻNE od 3!
Ta instrukcja zawsze zostanie wykonana
```

System.out.print kontra System.out.println

Jeśli byłeś uważny (w co nie wątpimy), na pewno zauważyłeś, 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 **print*newline***, 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 print**ln**. Jeśli natomiast łańcuchy mają być wyświetlane razem, należy się posłużyć metodą print.

Zaostrz 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");
  }
 }
}
```

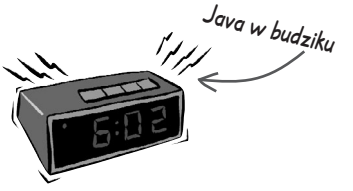
Tworzenie poważnej aplikacji biznesowej

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* spojrzysz na kod zamieszczony na tej stronie, zastanów się przez chwilę, w jaki sposób *samemu* napisałbyś klasyczną, ulubioną piosenkę dziecięcą 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";
      }

      System.out.println(iloscButelek + " " + slowo + " of beer on the
      wall"); System.out.println(iloscButelek + " " + slowo + " of
      beer."); System.out.println("Takie 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.

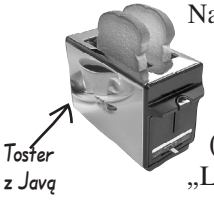
W poniedziałek rano u Roberta



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".

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".

Pięć 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 przerażony 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.

Tost Roberta jest upieczony.

Jego kawa paruje w filiżance.

A gazeta leży na miejscu.

Ot, kolejny poranek w ***domu obsługiwanym przez Javę***.

Także Ty możesz mieszkać w takim domu. Korzystaj z rozwiązań bazujących na języku Java, Ethernecie 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, pagery, karty inteligentne 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*.

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

W porządku, trzeba przyznać, że program piosenki o piwie nie był tak naprawdę poważną aplikacją biznesową. To był jedynie mały żart. Wciąż jednak musisz pokazać swojemu szefowi coś naprawdę poważnego? Przyjrzyj się zatem przedstawionemu kodowi programu krasomówczego. To cacko może Ci zapewnić dostęp do kluczy do łazienki kierowniczej.

Notatka: Wpisując kod programu w edytorze, pozwól, aby to sam tekst decydował, gdzie mają być dzielone wiersze i słowa! Nigdy nie naciskaj klawisza Enter, wpisując długi łańcuch znaków (czyli ciąg znaków zapisany pomiędzy znakami cudzysłowu), gdyż taki program się nie skompiluje. A zatem łączniki widoczne w przedstawionym kodzie są prawdziwe i można je umieścić w tekście, po ich wpisaniu nie należy jednak naciskać klawisza Enter. Można to zrobić dopiero po zamknięciu wpisywanego łańcucha znaków.

Tak naprawdę to nie wiemy, czym *jest* łazienka kierownicza. Jeśli Ty wiesz, prześlij nam proszę zdjęcie pocztą elektroniczną.

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

❶ // trzy grupy słów, które będą wybierane do zdania (dodaj własne!)

```
    String[] listaSlow1 = {"architektura wielowarstwowa", "30000 metrów", "rozwiązanie B-do-B", "aplikacja kliencka", "interfejs internetowy", "inteligentna karta", "rozwiązanie dynamiczne", "sześć sigma", "przenikliwość"};

    String[] listaSlow2 = {"zwiększa możliwości", "poprawia atrakcyjność", "pomnaża 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"};
```

❷ // określenie, ile jest słów w każdej z list

```
    int lista1Dlugosc = listaSlow1.length;
    int lista2Dlugosc = listaSlow2.length;
    int lista3Dlugosc = listaSlow3.length;
```

❸ // generacja trzech losowych słów (lub zwrotów)

```
    int rnd1 = (int) (Math.random() * lista1Dlugosc);
    int rnd2 = (int) (Math.random() * lista2Dlugosc);
    int rnd3 = (int) (Math.random() * lista3Dlugosc);
```

❹ // stworzenie zdania

```
    String zdanie = listaSlow1[rnd1] + " " + listaSlow2[rnd2] + " " + listaSlow3[rnd3];
```

❺ // 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 grzecznych 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 (miejsca 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):

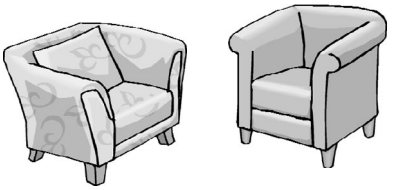
```
String s = zwierzaki[0]; // s ma wartość "Burek"
s = s + " " + "to pies"; // s ma wartość "Burek to pies"
```

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
- karta inteligentna podniesiona na wyższy poziom paradygmatu
- rozwiązanie B-do-B poprawia atrakcyjność portalu
- aplikacja kliencka opracowana dla rozwiązania
- sześć sigma skierowana na punkt wpisywania

Pogawędki przy kominku



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

Wirtualna 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 nie poprosi się mnie o **jego wykonanie**.

Kompilator

Nie podoba mi się ten ton.

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

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.

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 straciłbyś pracę, bratku.

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.

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

Wirtualna 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 — mógłby *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 metod, 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 wrócimy do tego tematu w kolejnej pogawędce.

Ćwiczenia

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

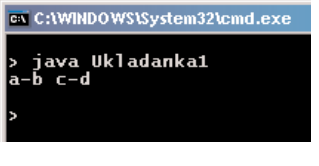
```
{  if (x > 2) {
   System.out.print("a");
}
```

```
int x = 3;
```

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

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

Wyniki:



Ćwiczenia

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 Cwiczenie1A {
  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 Cwiczenie1C {
    int x = 5;
    while (x > 1) {
      x = x - 1;
      if (x < 3) {
        System.out.println("Malutkie x");
      }
    }
}
```

Pomieszane 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 można 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

Połącz 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
```

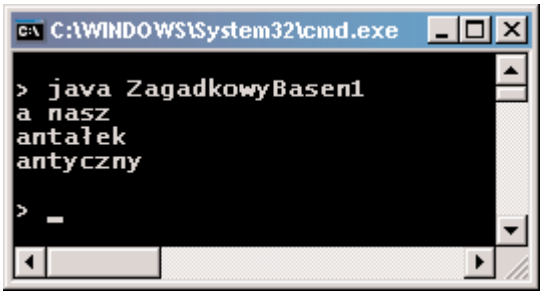
```
02 14 25 36 47
```

Zagadkowy basen

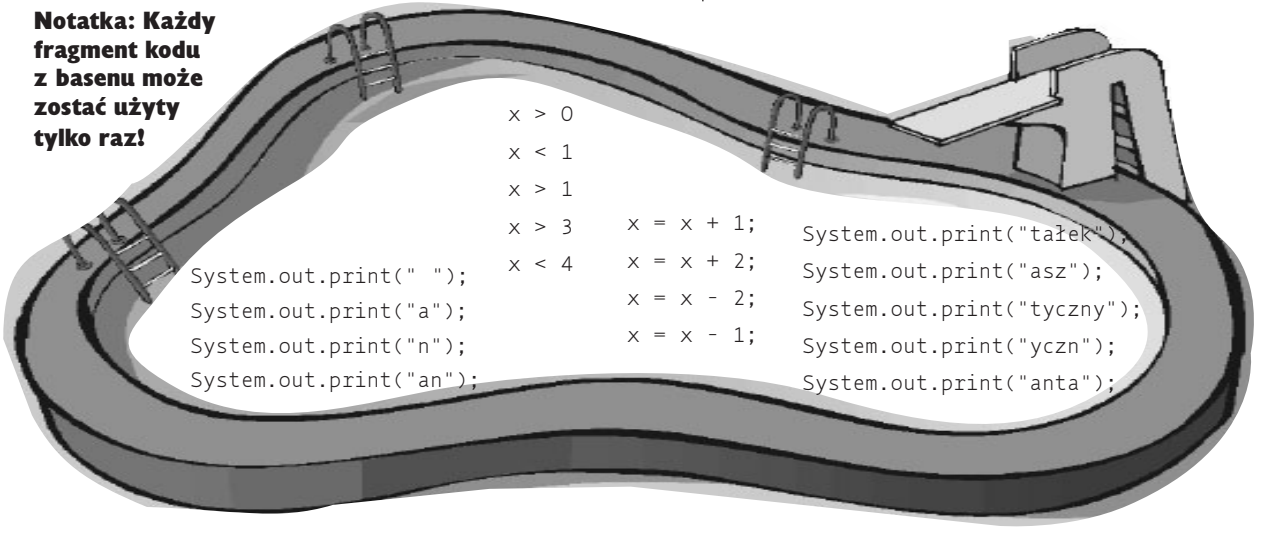
Twoim **zadaniem** jest wybranie fragmentów kodu z basenu i umieszczenie ich w miejscach kodu oznaczonych podkreśleniami. Żadnego fragmentu kodu nie można użyć więcej niż raz, a co więcej, nie wszystkie fragmenty zostaną wykorzystane. Zadanie polega na stworzeniu klasy, którą będzie można skompilować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

```
class ZagadkowyBasen1 {
  public static void main(String[] args)
{
    int x = 0;
    while ( _____ ) {
      ____________________________
      if (x < 1) {
        ____________________________
      }
      ____________________________
      if ( ______ ) {
        ____________________________
        _________
      }

      if (x == 1) {
        ____________________________
      }
      if ( _________ ) {
        ____________________________
      }

      System.out.println("");

      _____________
    }
  }
}
```

Wyniki:



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



```
x > 0
x < 1
x > 1
x > 3
x < 4

x = x + 1;
x = x + 2;
x = x - 2;
x = x - 1;

System.out.print("tałek");
System.out.print("asz");
System.out.print("tyczny");
System.out.print("yczn");
System.out.print("anta");

System.out.print(" ");
System.out.print("a");
System.out.print("n");
System.out.print("an");
```

Rozwiązania zadań

Magnesiki z kodem

```
class Ukladanka1 {
  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;
      }
    }
  }
}
```

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 należy zapominać 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");
      }
    }
  }
}
```

Klasa musi mieć przynajmniej jedną metodę (choć wcale nie musi to być metoda „main"!).

Odpowiedzi na zagadki

Gratulujemy! Udało Ci się dotrzeć do pierwszej strony z odpowiedziami w tej książce.

```
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("tałek");
      }
      if ( x < 1 ) {
        System.out.print("asz");
      }

      System.out.println("");

      x = x + 1;
    }
  }
}
```

```
C:\WINDOWS\System32\cmd.exe
> java ZagadkowyBasen1
a nasz
antałek
antyczny
```

```
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:

```
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
```