

ORACLE®

Java

Przewodnik dla początkujących

Wydanie VI

Herbert Schildt

Helion 

Oracle
Press™

Tytuł oryginału: Java™: A Beginner's Guide, Sixth Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-0808-4

Original edition copyright © 2014 by McGraw-Hill Education (Publisher).
All rights reserved.

Polish edition copyright © 2015 by HELION SA
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/javpp6.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/javpp6>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

O autorze	13
O redaktorze technicznym	14
Wstęp	15
1. Podstawy Javy	19
Pochodzenie Javy	20
Java a języki C i C++	20
Java a C#	21
Java a Internet	21
Aplety Java	21
Bezpieczeństwo	22
Przenośność	22
Magiczny kod bajtowy	22
Terminologia Javy	23
Programowanie obiektowe	24
Hermetyzacja	25
Polimorfizm	26
Dziedziczenie	26
Java Development Kit	26
Pierwszy prosty program	27
Wprowadzenie tekstu programu	28
Kompilowanie programu	28
Pierwszy program wiersz po wierszu	29
Obsługa błędów składni	31
Drugi prosty program	31
Inne typy danych	33
Dwie instrukcje sterujące	35
Instrukcja if	35
Pętla for	36
Bloki kodu	37
Średnik i pozycja kodu w wierszu	38
Wcięcia	39
Słowa kluczowe języka Java	41
Identyfikatory	41
Biblioteki klas	41
Test sprawdzający	42

4 Java. Przewodnik dla początkujących

2. Typy danych i operatory	43
Dlaczego typy danych są tak ważne	43
Typy proste	44
Typy całkowite	44
Typy zmiennoprzecinkowe	45
Znaki	46
Typ logiczny	47
Literały	49
Literały szesnastkowe, ósemkowe i binarne	49
Specjalne sekwencje znaków	49
Literały łańcuchowe	50
Zmienne	51
Inicjalizacja zmiennej	51
Dynamiczna inicjalizacja	51
Zasięg deklaracji i czas istnienia zmiennych	52
Operatory	54
Operatory arytmetyczne	54
Inkrementacja i dekrementacja	55
Operatory relacyjne i logiczne	56
Warunkowe operatory logiczne	58
Operator przypisania	59
Skrótowe operatory przypisania	59
Konwersje typów w instrukcjach przypisania	60
Rzutowanie typów niezgodnych	61
Priorytet operatorów	63
Wyrażenia	64
Konwersja typów w wyrażeniach	64
Odstępy i nawiasy	66
Test sprawdzający	66
3. Instrukcje sterujące	67
Wprowadzanie znaków z klawiatury	67
Instrukcja if	68
Zagnieżdżanie instrukcji if	69
Drabinka if-else-if	70
Instrukcja switch	71
Zagnieżdżanie instrukcji switch	74
Pętla for	76
Wariacje na temat pętli for	77
Brakujące elementy	78
Pętla nieskończona	79
Pętle bez ciała	79
Deklaracja zmiennych sterujących wewnątrz pętli	80
Rozszerzona pętla for	80
Pętla while	81
Pętla do-while	82
Przerywanie pętli instrukcją break	86
Zastosowanie break jako formy goto	87
Zastosowanie instrukcji continue	91
Pętle zagnieżdżone	94
Test sprawdzający	95

4. Wprowadzenie do klas, obiektów i metod	97
Podstawy klas	97
Ogólna postać klasy	98
Definiowanie klasy	98
Jak powstają obiekty	101
Referencje obiektów i operacje przypisania	101
Metody	102
Dodajemy metodę do klasy Vehicle	102
Powrót z metody	104
Zwracanie wartości	105
Stosowanie parametrów	106
Dodajemy sparametryzowaną metodę do klasy Vehicle	108
Konstruktory	113
Konstruktory z parametrami	114
Dodajemy konstruktor do klasy Vehicle	115
Operator new	116
Odzyskiwanie pamięci i metoda finalize()	116
Metoda finalize()	117
Słowo kluczowe this	119
Test sprawdzający	121
5. Więcej typów danych i operatorów	123
Tablice	123
Tablice jednowymiarowe	124
Tablice wielowymiarowe	128
Tablice dwuwymiarowe	128
Tablice nieregularne	129
Tablice o trzech i więcej wymiarach	130
Inicjalizacja tablic wielowymiarowych	130
Alternatywna składnia deklaracji tablic	131
Przypisywanie referencji tablic	131
Wykorzystanie składowej length	132
Styl for-each pętli for	137
Iteracje w tablicach wielowymiarowych	139
Zastosowania rozszerzonej pętli for	140
Łańcuchy znaków	141
Tworzenie łańcuchów	141
Operacje na łańcuchach	142
Tablice łańcuchów	144
Łańcuchy są niezmiennie	144
Łańcuchy sterujące instrukcją switch	145
Wykorzystanie argumentów wywołania programu	146
Operatory bitowe	147
Operatory bitowe AND, OR, XOR i NOT	147
Operatory przesunięcia	151
Skrótowe bitowe operatory przypisania	153
Operator ?	155
Test sprawdzający	157
6. Więcej o metodach i klasach	159
Kontrola dostępu do składowych klasy	159
Modyfikatory dostępu w Javie	160
Przekazywanie obiektów do metod	164
Sposób przekazywania argumentów	165

6 Java. Przewodnik dla początkujących

Zwracanie obiektów	167
Przeciążanie metod	169
Przeciążanie konstruktorów	173
Rekurencja	177
Słowo kluczowe static	178
Bloki static	181
Klasy zagnieżdżone i klasy wewnętrzne	184
Zmienne liczby argumentów	186
Metody o zmiennej liczbie argumentów	187
Przeciążanie metod o zmiennej liczbie argumentów	189
Zmienna liczba argumentów i niejednoznaczność	190
Test sprawdzający	191

7. Dziedziczenie 193

Podstawy dziedziczenia	193
Dostęp do składowych a dziedziczenie	196
Konstruktory i dziedziczenie	198
Użycie słowa kluczowego super do wywołania konstruktora klasy bazowej	199
Użycie słowa kluczowego super do dostępu do składowych klasy bazowej	203
Wielopoziomowe hierarchie klas	206
Kiedy wywoływane są konstruktory?	208
Referencje klasy bazowej i obiekty klasy pochodnej	209
Przesłanie metod	213
Przesłanie metod i polimorfizm	215
Po co przesłaniać metody?	216
Zastosowanie przesłaniania metod w klasie TwoDShape	217
Klasy abstrakcyjne	220
Słowo kluczowe final	223
final zapobiega przesłanianiu	223
final zapobiega dziedziczeniu	223
Użycie final dla zmiennych składowych	224
Klasa Object	225
Test sprawdzający	226

8. Pakiety i interfejsy 227

Pakiety	227
Definiowanie pakietu	228
Wyszukiwanie pakietów i zmienna CLASSPATH	228
Prosty przykład pakietu	229
Pakiety i dostęp do składowych	230
Przykład dostępu do pakietu	230
Składowe chronione	232
Import pakietów	234
Biblioteka klas Java używa pakietów	235
Interfejsy	235
Implementacje interfejsów	237
Referencje interfejsu	239
Zmienne w interfejsach	245
Interfejsy mogą dziedziczyć	246
Domyślne metody interfejsów	246
Podstawowe informacje o metodach domyślnych	247
Praktyczny przykład metody domyślnej	249
Problemy wielokrotnego dziedziczenia	250

Stosowanie metod statycznych w interfejsach	251
Ostatnie uwagi o pakietach i interfejsach	252
Test sprawdzający	252
9. Obsługa wyjątków	253
Hierarchia wyjątków	254
Podstawy obsługi wyjątków	254
Słowa kluczowe try i catch	254
Prosty przykład wyjątku	255
Konsekwencje nieprzechwycenia wyjątku	257
Wyjątki umożliwiają obsługę błędów	258
Użycie wielu klauzul catch	259
Przechwytywanie wyjątków klas pochodnych	259
Zagnieżdżanie bloków try	261
Generowanie wyjątku	262
Powtórne generowanie wyjątku	262
Klasa Throwable	263
Klauzula finally	265
Użycie klauzuli throws	266
Trzy ostatnio dodane możliwości wyjątków	267
Wyjątki wbudowane w Javę	268
Tworzenie klas pochodnych wyjątków	270
Test sprawdzający	274
10. Obsługa wejścia i wyjścia	275
Strumienie wejścia i wyjścia	276
Strumienie bajtowe i strumienie znakowe	276
Klasy strumieni bajtowych	276
Klasy strumieni znakowych	276
Strumienie predefiniowane	277
Używanie strumieni bajtowych	278
Odczyt wejścia konsoli	278
Zapis do wyjścia konsoli	280
Odczyt i zapis plików za pomocą strumieni bajtowych	281
Odczyt z pliku	281
Zapis w pliku	284
Automatyczne zamykanie pliku	285
Odczyt i zapis danych binarnych	288
Pliki o dostępie swobodnym	291
Strumienie znakowe	293
Odczyt konsoli za pomocą strumieni znakowych	293
Obsługa wyjścia konsoli za pomocą strumieni znakowych	297
Obsługa plików za pomocą strumieni znakowych	298
Klasa FileWriter	298
Klasa FileReader	299
Zastosowanie klas opakowujących do konwersji łańcuchów numerycznych	299
Test sprawdzający	307
11. Programowanie wielowątkowe	309
Podstawy wielowątkowości	309
Klasa Thread i interfejs Runnable	310
Tworzenie wątku	310
Drobne usprawnienia	313
Tworzenie wielu wątków	317
Jak ustalić, kiedy wątek zakończy działanie?	319

Priorytety wątków	321
Synchronizacja	324
Synchronizacja metod	324
Synchronizacja instrukcji	327
Komunikacja międzywątkowa	328
Przykład użycia metod wait() i notify()	329
Wstrzymywanie, wznowianie i kończenie działania wątków	333
Test sprawdzający	338

12. Typy wyliczeniowe, automatyczne opakowywanie, import składowych statycznych i adnotacje 339

Wyliczenia	340
Podstawy wyliczeń	340
Wyliczenia są klasami	342
Metody values() i valueOf()	342
Konstruktory, metody, zmienne instancji a wyliczenia	343
Dwa ważne ograniczenia	345
Typy wyliczeniowe dziedziczą po klasie Enum	345
Automatyczne opakowywanie	351
Typy opakowujące	351
Podstawy automatycznego opakowywania	352
Automatyczne opakowywanie i metody	353
Automatyczne opakowywanie i wyrażenia	354
Przeostroga	356
Import składowych statycznych	356
Adnotacje (metadane)	358
Test sprawdzający	361

13. Typy sparametryzowane 363

Podstawy typów sparametryzowanych	364
Prosty przykład typów sparametryzowanych	364
Parametryzacja dotyczy tylko typów obiektowych	367
Typy sparametryzowane różnią się dla różnych argumentów	367
Klasa sparametryzowana o dwóch parametrach	368
Ogólna postać klasy sparametryzowanej	369
Ograniczanie typów	369
Stosowanie argumentów wieloznacznych	372
Ograniczanie argumentów wieloznacznych	374
Metody sparametryzowane	376
Konstruktory sparametryzowane	378
Interfejsy sparametryzowane	378
Typy surowe i tradycyjny kod	384
Wnioskowanie typów i operator diamentowy	387
Wymazywanie	388
Błędy niejednoznaczności	388
Ograniczenia związane z typami sparametryzowanymi	389
Zakaz tworzenia instancji parametru typu	389
Ograniczenia dla składowych statycznych	389
Ograniczenia tablic sparametryzowanych	390
Ograniczenia związane z wyjątkami	391
Dalsze studiowanie typów sparametryzowanych	391
Test sprawdzający	391

14. Wyrażenia lambda i referencje metod	393
Przedstawienie wyrażen lambda	394
Podstawowe informacje o wyrażeniach lambda	394
Interfejsy funkcyjne	395
Wyrażenia lambda w działaniu	397
Blokowe wyrażenia lambda	400
Sparametryzowane interfejsy funkcyjne	401
Wyrażenia lambda i przechwytywanie zmiennych	407
Zgłaszanie wyjątków w wyrażeniach lambda	408
Referencje metod	409
Referencje metod statycznych	409
Referencje metod instancyjnych	411
Referencje konstruktorów	414
Predefiniowane interfejsy funkcyjne	417
Test sprawdzający	418
15. Aplety, zdarzenia i pozostałe słowa kluczowe	421
Podstawy apletów	422
Organizacja apletów i podstawowe elementy	424
Architektura apletu	424
Kompletny szkielet apletu	425
Rozpoczęcie i zakończenie działania apletu	426
Żądanie odrysowania	426
Metoda update()	427
Wykorzystanie okna statusu	430
Parametry apletów	431
Klasa Applet	432
Obsługa zdarzeń	433
Model delegacji zdarzeń	433
Zdarzenia	435
Źródła zdarzeń	435
Obiekty nasłuchujące	435
Klasy zdarzeń	435
Interfejsy obiektów nasłuchujących	436
Wykorzystanie modelu delegacji zdarzeń	436
Obsługa zdarzeń myszy	437
Prosty aplet obsługujący zdarzenia myszy	438
Inne słowa kluczowe Javy	440
Modyfikatory transient i volatile	440
Operator instanceof	441
Słowo kluczowe strictfp	441
Słowo kluczowe assert	441
Metody natywne	442
Test sprawdzający	443
16. Wprowadzenie do Swing	445
Pochodzenie i filozofia Swing	446
Komponenty i kontenery	447
Komponenty	448
Kontenery	448
Panele kontenerów szczytowych	448
Menedżery układu	449
Pierwszy program wykorzystujący Swing	450
Pierwszy program Swing wiersz po wierszu	451

10 Java. Przewodnik dla początkujących

Komponent JButton	454
Komponent JTextField	457
Komponent JCheckBox	459
Komponent JList	462
Wykorzystanie anonimowych klas wewnętrznych lub wyrażeń lambda do obsługi zdarzeń	470
Aplety Swing	471
Test sprawdzający	473
17. Wprowadzenie do JavaFX	475
Podstawowe pojęcia JavaFX	476
Pakiety JavaFX	476
Klasy Stage oraz Scene	476
Węzły i graf sceny	477
Układy	477
Klasa Application oraz metody cyklu życia	477
Uruchamianie aplikacji JavaFX	478
Szkielet aplikacji JavaFX	478
Kompilacja i uruchamianie programów JavaFX	481
Wątek aplikacji	481
Prosta kontrolka JavaFX: Label	481
Stosowanie przycisków i zdarzeń	483
Podstawy obsługi zdarzeń	484
Przedstawienie kontrolki Button	484
Przedstawienie obsługi zdarzeń i stosowania przycisków	485
Trzy kolejne kontrolki JavaFX	487
Pola wyboru	487
Listy	491
Pola tekstowe	495
Przedstawienie efektów i transformacji	498
Efekty	498
Transformacje	500
Prezentacja zastosowania efektów i transformacji	501
Co dalej?	503
Test sprawdzający	504
A Rozwiązania testów sprawdzających	505
Rozdział 1. Podstawy Javy	505
Rozdział 2. Typy danych i operatory	507
Rozdział 3. Instrukcje sterujące	508
Rozdział 4. Wprowadzenie do klas, obiektów i metod	510
Rozdział 5. Więcej typów danych i operatorów	511
Rozdział 6. Więcej o metodach i klasach	514
Rozdział 7. Dziedziczenie	518
Rozdział 8. Pakiety i interfejsy	519
Rozdział 9. Obsługa wyjątków	521
Rozdział 10. Obsługa wejścia i wyjścia	523
Rozdział 11. Programowanie wielowątkowe	526
Rozdział 12. Typy wyliczeniowe, automatyczne opakowywanie, import składowych statycznych i adnotacje	528
Rozdział 13. Typy sparametryzowane	531
Rozdział 14. Wyrażenia lambda i referencje metod	534
Rozdział 15. Aplety, zdarzenia i pozostałe słowa kluczowe	537
Rozdział 16. Wprowadzenie do Swing	541
Rozdział 17. Wprowadzenie do JavaFX	546

B	Komentarze dokumentacyjne	551
	Znaczniki javadoc	551
	@author	552
	{@code}	552
	@deprecated	552
	{@docRoot}	553
	@exception	553
	{@inheritDoc}	553
	{@link}	553
	{@linkplain}	553
	{@literal}	553
	@param	553
	@return	553
	@see	554
	@serial	554
	@serialData	554
	@serialField	554
	@since	554
	@throws	554
	{@value}	554
	@version	555
	Ogólna postać komentarza dokumentacyjnego	555
	Wynik działania programu javadoc	555
	Przykład użycia komentarzy dokumentacyjnych	555
	 Skorowidz	 557

Pakiety i interfejsy

W tym rozdziale poznasz:

- stosowanie pakietów,
- wpływ pakietów na poziom dostępu,
- modyfikator dostępu `protected`,
- importowanie pakietów,
- standardowe pakiety Javy,
- podstawy interfejsów,
- implementacje interfejsów,
- referencje interfejsów,
- zmienne interfejsów,
- dziedziczenie interfejsów,
- domyślne i statyczne metody interfejsów.

Ten rozdział poświęcę na omówienie dwóch najbardziej innowacyjnych elementów Javy: pakietów i interfejsów. **Pakiety** są grupami powiązanych klas. Pakiety pomagają organizować kod i tworzą kolejną warstwę hermetyzacji. **Interfejs** definiuje zbiór metod, które zostaną zaimplementowane przez klasę. A zatem interfejs zapewnia możliwość określenia, co klasa będzie robić, lecz nie tego, jak to zrobi. Znajomość pakietów i interfejsów pozwoli Ci lepiej kontrolować organizację kodu.

Pakiety

W programowaniu pomocna jest możliwość grupowania powiązanych części programu. W Javie służą do tego pakiety. Pakiet spełnia dwa zadania. Po pierwsze, zapewnia mechanizm umożliwiający zorganizowanie powiązanych elementów programu. Dostęp do klas zdefiniowanych wewnątrz pakietu odbywa się z użyciem nazwy pakietu. W ten sposób pakiet dostarcza nazwy dla kolekcji klas. Po drugie, pakiet jest częścią mechanizmu kontroli dostępu stosowanego w Javie. Klasy zdefiniowane w pakiecie mogą być prywatne dla tego pakietu i tym samym niedostępne na zewnątrz pakietu. W ten sposób pakiet umożliwia hermetyzację klas. Przyjrzyjmy się bliżej każdej z wymienionych możliwości.

Tworząc klasę, przydzielamy jej nazwę należącą do **przestrzeni nazw**. Przestrzeń nazw definiuje region deklaracji. W Javie żadne dwie klasy nie mogą używać tej samej nazwy w tej samej przestrzeni nazw. Innymi słowy, nazwa klasy musi być unikatowa w danej przestrzeni nazw. Wszystkie dotychczasowe przykłady używały domyślnej (globalnej) przestrzeni nazw. Rozwiązanie takie sprawdza się w przypadku prostych programów, ale staje się problematyczne, gdy złożoność programu wzrasta

i w domyślnej przestrzeni nazw staje się tłoczno. W rozbudowanych programach nadanie klasie unikatowej nazwy może wtedy być problemem. Co gorsza, musisz również unikać kolizji nazw z kodem tworzonym przez innych programistów w ramach tego samego projektu, a także z bibliotekami Javy. Rozwiązanie tego problemu jest możliwe dzięki użyciu pakietu, gdyż umożliwia on podział przestrzeni nazw. Jeśli zdefiniujesz klasę w pakiecie, to do jej nazwy dołączasz nazwę pakietu i w ten sposób unikasz kolizji z innymi klasami o tej samej nazwie, ale należącymi do innych pakietów.

Ponieważ pakiety zawierają zwykle powiązane klasy, Java definiuje specjalne prawa dostępu do kodu umieszczonego w pakiecie. W pakiecie możesz zdefiniować kod, który jest dostępny dla pozostałego kodu w tym samym pakiecie, ale nie na zewnątrz pakietu. Pozwala to tworzyć samodzielne grupy powiązanych klas, których działanie pozostaje prywatne.

Definiowanie pakietu

Wszystkie klasy w Javie należą do jakiegoś pakietu. Jeśli nie użyjesz jawnie instrukcji `package`, to klasa należy do domyślnego (globalnego) pakietu. Pakiet domyślny nie ma nazwy i wobec tego jego użycie jest transparentne. Z tego powodu jak dotąd nie musiałeś się przejmować nazwami pakietów. Pakiet domyślny sprawdza się w przypadku prostych programów, ale nie w przypadku prawdziwych aplikacji. Tworząc te ostatnie, będziesz zwykle definiować jeden lub więcej pakietów.

Aby stworzyć pakiet, musisz umieścić instrukcję `package` na początku pliku źródłowego. Klasy definiowane w tym pliku będą wtedy należeć do tego pakietu. Ponieważ pakiet definiuje przestrzeń nazw, nazwy klas zdefiniowanych w tym pliku należą do przestrzeni nazw pakietu.

A oto ogólna postać instrukcji `package`:

```
package nazwa-pakietu;
```

Poniższa instrukcja tworzy pakiet o nazwie `mypack`:

```
package mypack;
```

Java używa systemu plików do zarządzania pakietami, umieszczając każdy pakiet we własnym katalogu. Na przykład pliki `.class` dla wszystkich klas zdefiniowanych w pakiecie `mypack` muszą zostać umieszczone w katalogu o nazwie `mypack`.

Podobnie jak dla innych elementów języka Java, również w przypadku nazw pakietów rozróżniane są małe i duże litery. Musisz zatem pamiętać, aby nazwa katalogu, w którym umieszczasz pliki klas, była dokładnie taka sama jak nazwa pakietu. Jeśli podczas uruchamiania przykładów zamieszczonych w tym rozdziale napotkasz problemy, sprawdź dokładnie nazwy katalogów i pakietów. Z tego powodu programiści zwykle używają jedynie małych liter w nazwach pakietów.

Tę samą instrukcję `package` możesz umieścić w więcej niż jednym pliku. Instrukcja `package` określa po prostu, do jakiego pakietu należą klasy zdefiniowane w pliku. Nie wyklucza zatem możliwości, aby inne klasy w innych plikach należały do tego samego pakietu. I rzeczywiście, w praktyce większość pakietów zwykle składa się z wielu plików.

Istnieje możliwość tworzenia hierarchii pakietów. W tym celu nazwy pakietów kolejnych poziomów oddzielamy od siebie kropką. Ogólna postać instrukcji `package` dotyczącej wielopoziomowej hierarchii pakietów jest następująca:

```
package pakiet1.pakiet2.pakiet3...pakietN;
```

W takim przypadku musisz zadbać o stworzenie odpowiedniej struktury katalogów. Na przykład klasy pakietu

```
package alpha.beta.gamma;
```

muszą zostać umieszczone w katalogu `.../alpha/beta/gamma`, gdzie `...` oznacza ścieżkę dostępu do tego katalogu.

Wyszukiwanie pakietów i zmienna CLASSPATH

Jak już wyjaśniłem, pakietom odpowiadają katalogi. Powstaje zatem pytanie: skąd Java „wie”, gdzie szukać utworzonych przez Ciebie pakietów? Odpowiedź na to pytanie składa się z trzech części. Po pierwsze, Java używa jako punktu wyjścia bieżącego katalogu. Zatem jeśli umieściłeś pakiet w pod-

katalogu bieżącego katalogu, zostanie on odnaleziony. Po drugie, możesz określić ścieżkę dostępu do katalogu za pomocą zmiennej środowiskowej CLASSPATH. Po trzecie, możesz użyć w tym samym celu opcji `-classpath`, wywołując programy `javac` i `java`.

Na przykład zakładając poniższą specyfikację pakietu:

```
package mypack;
```

aby pakiet `mypack` został odnaleziony, musi zostać spełniony jeden z następujących warunków: program musi być uruchamiany z katalogu znajdującego się bezpośrednio nad katalogiem `mypack`, zmienna CLASSPATH musi zawierać ścieżkę dostępu do katalogu `mypack` lub podczas uruchamiania interpretera `java` musisz określić ścieżkę dostępu do katalogu `mypack`, używając opcji `-classpath`.

Najprostszy sposób uruchomienia przykładów zamieszczonych w tej książce polega na utworzeniu podkatalogów dla pakietów poniżej bieżącego katalogu, umieszczeniu plików `.class` w odpowiednich podkatalogach i uruchamianiu programów z katalogu bieżącego. Rozwiązanie to zastosuję w kolejnych przykładach.

I jeszcze jedno: aby uniknąć problemów, najlepiej przechowywać wszystkie pliki `.java` i `.class` związane z danym pakietem w jego podkatalogu. A także kompilować każdy plik z poziomu katalogu bieżącego znajdującego się nad podkatalogiem pakietu.

Prosty przykład pakietu

Spróbujmy zastosować w praktyce informacje z poprzedniego podrozdziału. Przykład przedstawiony na listingu 8.1 tworzy prostą bazę danych zawierającą informacje o książkach, która należy do pakietu `bookpack`.

Listing 8.1. *BookDemo.java*

```
// Demonstracja prostego pakietu.
package bookpack; ← Plik należy do pakietu bookpack.

class Book { ← Zatem klasa Book również należy do bookpack.
    private String title;
    private String author;
    private int pubDate;

    Book(String t, String a, int d) {
        title = t;
        author = a;
        pubDate = d;
    }

    void show() {
        System.out.println(title);
        System.out.println(author);
        System.out.println(pubDate);
        System.out.println();
    }
}

class BookDemo { ← Klasa BookDemo również należy do bookpack.
    public static void main(String args[]) {
        Book books[] = new Book[5];

        books[0] = new Book("Java. Przewodnik dla początkujących",
                            "Schildt", 2015);
        books[1] = new Book("Java. Kompendium programisty",
                            "Schildt", 2012);
        books[2] = new Book("Pan Tadeusz",
                            "Mickiewicz", 1974);
        books[3] = new Book("Stan zagrożenia",
                            "Clancy", 2003);
        books[4] = new Book("Ogniem i mieczem",
```

```

        "Sienkiewicz", 1995);
    for(int i=0; i < books.length; i++) books[i].show();
}
}

```

Nazwij plik źródłowy *BookDemo.java* i umieść go w katalogu *bookpack*.
Następnie skompiluj program. W tym celu możesz wywołać

```
javac bookpack/BookDemo.java
```

z poziomu katalogu znajdującego się bezpośrednio nad katalogiem *bookpack*. Spróbuj uruchomić program za pomocą poniższego polecenia:

```
java -Dfile.encoding=CP852 bookpack.BookDemo
```

Pamiętaj, że podczas uruchamiania programu katalogiem bieżącym powinien być katalog bezpośredni nad katalogiem *bookpack* (lub użyj jednej z dwóch możliwości specyfikacji dostępu do katalogu *bookpack* omówionych w poprzednim podrozdziale).

Klasy *BookDemo* i *Book* należą teraz do pakietu *bookpack*. Oznacza to, że nie możesz teraz wykonać kodu klasy *BookDemo*. Poniższe polecenie nie powiedzie się:

```
java -Dfile.encoding=CP852 BookDemo
```

Odwołując się do klasy *BookDemo*, musisz podać jej pełną nazwę wraz z nazwą pakietu.

Pakiety i dostęp do składowych

We wcześniejszych rozdziałach omówiłem podstawy kontroli dostępu w Javie, w tym modyfikatory `private` i `public`. Nie wyczerpuje to jednak zagadnienia. Pakiety również uczestniczą w kontroli dostępu i dlatego kompletne omówienie zagadnienia musiało poczekać, aż przedstawię pakiety.

Dostępność elementu jest określona przez specyfikację dostępu, domyślną, `private`, `public` lub `protected`. A także przez pakiet, do którego należy element. Zatem dostępność elementu jest określona przez jego dostępność wewnątrz klasy i dostępność wewnątrz pakietu. Takie wielowarstwowe rozwiązanie kontroli dostępu umożliwia wprowadzenie bogatego zbioru uprawnień dostępu. Różne poziomy dostępu przedstawiłem w tabeli 8.1. Przeanalizujmy każdą z opcji dostępu z osobna.

Jeśli składowa klasy nie ma jawnego modyfikatora dostępu, to jest dostępna wewnątrz swojego pakietu, ale nie na zewnątrz pakietu. Dlatego domyślny poziom dostępu stosujemy dla elementów, które mają być publicznie dostępne wewnątrz pakietu, ale pozostawać prywatne dla tego pakietu.

Składowe zadeklarowane jawnie jako `public` (nazywane także składowymi publicznymi) są dostępne wszędzie, w różnych klasach i różnych pakietach. Nie ma żadnych restrykcji związanych z dostępem do tych składowych i ich użyciem. Składowa zadeklarowana jako `private` (nazywana także składową prywatną) jest dostępna wyłącznie dla innych składowych tej samej klasy. Zatem przynależność do pakietu nie ma żadnego wpływu na dostęp do składowej prywatnej. Składowa zadeklarowana jako `protected` (nazywana także składową chronioną) jest dostępna wewnątrz swojego pakietu oraz dla wszystkich klas pochodnych, z klasami pochodnymi należącymi do innych pakietów włącznie.

Informacje podane w tabeli 8.1 dotyczą tylko składowych klas. W przypadku klasy istnieją tylko dwa poziomy dostępu: domyślny i `public`. Klasa zadeklarowana jako `public` jest dostępna dla dowolnego kodu. Jeśli natomiast ma domyślny poziom dostępu, jest dostępna jedynie dla kodu wewnątrz tego samego pakietu. Klasa zadeklarowana jako `public` musi rezydować w pliku o takiej samej nazwie jak nazwa klasy.

Przykład dostępu do pakietu

W poprzednim przykładzie klasy *Book* i *BookDemo* zostały umieszczone w tym samym pakiecie, a zatem klasa *BookDemo* mogła używać klasy *Book*, ponieważ domyślny poziom dostępu umożliwia swobodny dostęp wewnątrz pakietu. Sytuacja zmieniałaby się, gdyby klasy należały do różnych pakietów. W takim przypadku klasa *BookDemo* straciłaby dostęp do klasy *Book*. Aby udostępnić klasę *Book* innym pakietom,

Tabela 8.1. Dostępność składowych klasy

	Składowa prywatna	Składowa o domyślnym poziomie dostępu	Składowa chronina	Składowa publiczna
Dostępna w tej samej klasie	tak	tak	tak	tak
Dostępna w tym samym pakiecie dla klasy pochodnej	nie	tak	tak	tak
Dostępna w tym samym pakiecie dla klasy, która nie jest pochodną jej klasy	nie	tak	tak	tak
Dostępna w innym pakiecie dla klasy pochodnej	nie	nie	tak	tak
Dostępna w innym pakiecie dla klasy, która nie jest pochodną jej klasy	nie	nie	nie	tak

musisz wykonać trzy zmiany. Po pierwsze, zadeklarować klasę `Book` jako `public`. Dzięki temu klasa `Book` będzie dostępna na zewnątrz pakietu. Po drugie, jej konstruktor również musi być zadeklarowany jako `public`. I po trzecie, tak samo musisz postąpić z metodą `show()`. W ten sposób konstruktor i metoda `show()` będą dostępne również na zewnątrz pakietu. Zatem aby umożliwić wykorzystanie klasy `Book` przez inne pakiety, należy zdefiniować ją w sposób przedstawiony na listingu 8.2.

Listing 8.2. *Book.java*

```
// Klasa Book upubliczniona dla innych pakietów.
package bookpack;

public class Book { ←————— Klasa Book i jej składowe muszą być zadeklarowane jako public,
    private String title;          aby mogły zostać użyte w innych pakietach.
    private String author;
    private int pubDate;

    // Teraz zadeklarowany jako public.
    public Book(String t, String a, int d) {
        title = t;
        author = a;
        pubDate = d;
    }

    // Teraz zadeklarowana jako public.
    public void show() {
        System.out.println(title);
        System.out.println(author);
        System.out.println(pubDate);
        System.out.println();
    }
}
```

Aby użyć klasy `Book` w innym pakiecie, musisz użyć instrukcji `import` omawianej w następnym rozdziale lub podać nazwę klasy wraz z pełną nazwą pakietu, do którego należy. Na listingu 8.3 przedstawiłem klasę `UseBook` należącą do pakietu `bookpackext`. Używa ona klasy `Book`, podając jej nazwę wraz z nazwą pakietu.

Listing 8.3. *Usebook.java*

```
// Ta klasa należy do pakietu bookpackext.
package bookpackext;

// Używa klasy Book należącej do pakietu bookpack.
```

```

class UseBook {
    public static void main(String args[]) {
        backpack.Book books[] = new backpack.Book[5]; ← Odwołanie do klasy Book musisz
                                                         poprzedzić nazwą pakietu: backpack.

        books[0] = new backpack.Book("Java. Przewodnik dla początkujących",
                                     "Schildt", 2015);
        books[1] = new backpack.Book("Java. Kompendium programisty",
                                     "Schildt", 2012);
        books[2] = new backpack.Book("Pan Tadeusz",
                                     "Mickiewicz", 1974);
        books[3] = new backpack.Book("Stan zagrożenia",
                                     "Clancy", 2003);
        books[4] = new backpack.Book("Ogniem i mieczem",
                                     "Sienkiewicz", 1995);

        for(int i=0; i < books.length; i++) books[i].show();
    }
}

```

Zwróć uwagę, że wszystkie odwołania do klasy Book zostały poprzedzone nazwą pakietu backpack. Jeśli tego nie zrobisz, klasa Book nie zostanie odnaleziona podczas próby kompilacji klasy UseBook.

Składowe chronione

Rozpoczynającym przygodę z językiem Java zrozumienie znaczenia i zastosowania modyfikatora `protected` często sprawia kłopoty. Jak już wyjaśniłem, składowa chroniona, czyli składowa zadeklarowana jako `protected`, jest dostępna w swoim pakiecie oraz we wszystkich klasach pochodnych należących do innych pakietów. Zatem składowa chroniona jest dostępna dla wszystkich klas pochodnych, ale nie jest dostępna dla dowolnego kodu poza pakietem.

Aby lepiej wyjaśnić efekt zastosowanie modyfikatora `protected`, posłużę się przykładem. Najpierw zmodyfikujemy klasę Book w taki sposób, aby jej zmienne składowe były zadeklarowane jako chronione. Zmodyfikowaną wersję klasy Book przedstawiłem na listingu 8.4.

Listing 8.4. *Book.java*

// Zmienne składowe klasy Book zadeklarowane jako chronione.

```

package backpack;

public class Book {
    // teraz zadeklarowane jako protected
    protected String title;
    protected String author;
    protected int pubDate;
}

public Book(String t, String a, int d) {
    title = t;
    author = a;
    pubDate = d;
}

public void show() {
    System.out.println(title);
    System.out.println(author);
    System.out.println(pubDate);
    System.out.println();
}
}

```

Te składowe są teraz zadeklarowane jako chronione.

Następnie utwórz klasę pochodną klasy Book o nazwie ExtBook i klasę o nazwie ProtectDemo, która używa klasy ExtBook. Klasa ExtBook wprowadza dodatkową składową przechowującą nazwę wydawcy

i kilka metod dostępowych. Obie klasy umieścisz we własnym pakiecie o nazwie bookpackext. Przedstawiłem je na listingu 8.5.

Listing 8.5. *ProtectDemo.java*

```
// Demonstruje użycie modyfikatora protected.
package bookpackext;

class ExtBook extends bookpack.Book {
    private String publisher;

    public ExtBook(String t, String a, int d, String p) {
        super(t, a, d);
        publisher = p;
    }

    public void show() {
        super.show();
        System.out.println(publisher);
        System.out.println();
    }

    public String getPublisher() { return publisher; }
    public void setPublisher(String p) { publisher = p; }

    /* Poniższe metody są poprawne, gdyż klasa
       pochodna ma dostęp do składowych chronionych. */
    public String getTitle() { return title; }
    public void setTitle(String t) { title = t; }
    public String getAuthor() { return author; } ← Dostęp do składowych klasy Book
    public void setAuthor(String a) { author = a; }      jest dozwolony w klasach pochodnych.
    public int getPubDate() { return pubDate; }
    public void setPubDate(int d) { pubDate = d; }
}

class ProtectDemo {
    public static void main(String args[]) {
        ExtBook books[] = new ExtBook[5];

        books[0] = new ExtBook("Java. Przewodnik dla początkujących",
                               "Schildt", 2015, "Helion");
        books[1] = new ExtBook("Java. Kompendium programisty",
                               "Schildt", 2012, "Helion");
        books[2] = new ExtBook("Pan Tadeusz",
                               "Mickiewicz", 1974, "PIW");
        books[3] = new ExtBook("Stan zagrożenia",
                               "Clancy", 2003, "Amber");
        books[4] = new ExtBook("Ogniem i mieczem",
                               "Sienkiewicz", 1995, "PIW");

        for(int i=0; i < books.length; i++) books[i].show();

        // Wyszukuje książki konkretnego autora
        System.out.println("Wszystkie tytuły, których autorem jest Schildt.");
        for(int i=0; i < books.length; i++)
            if(books[i].getAuthor() == "Schildt")
                System.out.println(books[i].getTitle());

        // books[0].title = "test title"; // Błąd -- składowa nie jest tutaj dostępna
    }
}
```

Najpierw przyjrzyj się klasie ExtBook. Ponieważ jest ona klasą pochodną klasy Book, ma dostęp do chronionych składowych klasy Book, mimo że należy do innego pakietu. Może zatem bezpośrednio używać składowych `title`, `author` i `pubDate`, implementując metody dostępowe dla tych zmiennych. Jednak w klasie ProtectDemo dostęp do tych zmiennych nie jest możliwy, ponieważ ProtectDemo nie jest klasą pochodną klasy Book. Jeśli usuniesz znak komentarza rozpoczynający poniższy wiersz, to kompilacja programu zakończy się błędem.

```
// books[0].title = "test title"; // Błąd -- składowa nie jest tutaj dostępna
```

Import pakietów

W poprzednich przykładach pokazałem, że używając klas z innego pakietu, możesz poprzedzać ich nazwy pełną nazwą pakietu. Jednak takie rozwiązanie szybko staje się męczące i mało eleganckie, zwłaszcza gdy klasy należą do pakietu umieszczonego głęboko w hierarchii. Skoro Javę wymyślili programiści dla programistów, a programiści nie lubią pracochłonnej konstrukcji, nie będzie niespodzianką, że istnieje wygodniejszy sposób korzystania z zawartości pakietów: instrukcja `import`. Stosując tę instrukcję, możesz udostępnić jeden lub więcej elementów pakietu. Wtedy możesz ich używać bez podawania pełnych nazw pakietów.

Ekspert odpowiada

Pytanie: Również w języku C++ istnieje słowo kluczowe `protected`. Czy jego zastosowanie jest podobne jak w Javie?

Odpowiedź: Podobne, ale nie takie samo. W języku C++ składowa chroniona jest dostępna w klasach pochodnych, ale poza tym jest prywatna. W języku Java składowa chroniona jest dostępna dla dowolnego kodu w tym samym pakiecie, ale tylko dla klas pochodnych na zewnątrz pakietu. Zatem przepisując kod z języka C++ na Javę, musisz zwrócić uwagę na tę różnicę.

Ogólna postać instrukcji `import` jest następująca:

```
import pakiet.nazwaklasy
```

W tym przypadku *pakiet* jest pełną nazwą pakietu, a *nazwaklasy* dotyczy importowanej klasy. Jeśli chcesz importować całą zawartość pakietu, zamiast nazwy klasy używasz gwiazdki. Oto przykłady obu postaci instrukcji `import`:

```
import mypack.MyClass
import mypack.*;
```

W pierwszym przykładzie klasa `MyClass` zostaje importowana z pakietu `mypack`. Drugi przykład oznacza import wszystkich klas należących do pakietu `mypack`. W pliku źródłowym Java instrukcja `import` musi znaleźć się zaraz po instrukcji `package` (jeśli taka istnieje) i przed definicją jakiegokolwiek klasy.

Instrukcji `import` możemy użyć, aby udostępnić pakiet `bookpack` i używać klasy `Book` bez poprzedzania jej nazwą pakietu. Wystarczy w tym celu dodać poniższą instrukcję `import` na początku każdego pliku, który używa klasy `Book`.

```
import bookpack.*;
```

Na listingu 8.6 przedstawiłem wersję klasy `UseBook` wykorzystującą instrukcję `import`.

Listing 8.6. UseBook2.java

```
// Demonstruje instrukcję import.
package bookpackext;
import bookpack.*; ← Importuje cały pakiet bookpack.

// Używa klasy Book należącej do pakietu bookpack.
class UseBook2 {
    public static void main(String args[]) {
        Book books[] = new Book[5]; ← Teraz możesz odwoływać się do klasy Book bezpośrednio,
                                     bez nazwy pakietu.
```

```

books[0] = new Book("Java. Przewodnik dla początkujących",
                  "Schildt", 2015);
books[1] = new Book("Java. Kompendium programisty",
                  "Schildt", 2012);
books[2] = new Book("Pan Tadeusz",
                  "Mickiewicz", 1974);
books[3] = new Book("Stan zagrożenia",
                  "Clancy", 2003);
books[4] = new Book("Ogniem i mieczem",
                  "Sienkiewicz", 1995);

for(int i=0; i < books.length; i++) books[i].show();
}
}

```

Zwróć uwagę, że nazwa pakietu nie jest już używana dla klasy Book.

Biblioteka klas Java używa pakietów

Jak już wspomniałem w tej książce, Java definiuje wiele klas standardowych dostępnych dla wszystkich programów. Tę bibliotekę klas często określa się mianem Java API (*Application Programming Interface*). Klasy Java API zostały umieszczone w pakietach. Na szczycie ich hierarchii umieszczono pakiet `java`. Poniżej znajduje się wiele innych pakietów; kilka z nich przedstawiłem w tabeli 8.2.

Tabela 8.2. Pakiety Java API

Pakiet	Opis
<code>java.lang</code>	Zawiera wiele klas ogólnego przeznaczenia
<code>java.io</code>	Zawiera klasy obsługi wejścia i wyjścia
<code>java.net</code>	Zawiera klasy obsługi sieci
<code>java.applet</code>	Zawiera klasy umożliwiające tworzenie apletów
<code>java.awt</code>	Zawiera klasy tzw. Abstract Window Toolkit

Od początku tej książki korzystasz już z pakietu `java.lang`. Zawiera on między innymi klasę `System`, której używałeś, wyświetlając informacje za pomocą metody `println()`. Pakiet `java.lang` jest unikatowy pod tym względem, że jest automatycznie importowany do każdego programu w języku Java. Dlatego w dotychczasowych przykładach nie musiałeś używać instrukcji `import java.lang`. Pozostałe pakiety wymagają jednak jawnego użycia instrukcji `import`. Kilka z nich poznasz bliżej w kolejnych rozdziałach.

Interfejsy

W programowaniu obiektowym często pomocna jest możliwość zdefiniowania operacji, które musi wykonywać klasa, jednak bez określania szczegółów ich implementacji. Poznałeś już jeden przykład takiego rozwiązania: metodę abstrakcyjną. Metoda abstrakcyjna definiuje sygnaturę metody, ale nie dostarcza jej implementacji. Klasy pochodne muszą dostarczyć implementacji każdej metody abstrakcyjnej zdefiniowanej w klasie bazowej. Innymi słowy, metoda abstrakcyjna definiuje *interfejs*, ale nie *implementację*. Mimo że klasy abstrakcyjne i metody abstrakcyjne są przydatne, w Javie zdecydowano rozwinąć tę koncepcję o krok dalej. Udostępniono możliwość zupełnego oddzielenia interfejsu klasy od jej implementacji poprzez zastosowanie słowa kluczowego `interface`.

Pod względem składni interfejsy przypominają nieco klasy abstrakcyjne, gdyż także w nich można podać jedną lub więcej klas, które nie mają ciała. Aby czynności wykonywane przez te metody zostały zdefiniowane, wszystkie metody interfejsu należy zaimplementować. A zatem interfejsy określają, co musi być zrobione, ale nie określają, jak te czynności należy wykonać. Po zdefiniowaniu interfejsu może być implementowany przez dowolną liczbę klas. Jedna klasa może natomiast implementować dowolnie wiele interfejsów.

Implementacja interfejsu oznacza, że klasa musi dostarczyć ciał (implementacji) dla metod opisanych przez interfejs. Szczegóły implementacji są wewnętrzną sprawą klasy. Dwie klasy mogą implementować ten sam interfejs w różny sposób, ale każda z nich musi obsługiwać ten sam zbiór metod. Dzięki temu kod może za pomocą tego samego interfejsu używać obiektów obydwu klas. Udostępniając słowo kluczowe `interface`, Java umożliwia pełne wykorzystanie polimorfizmu rozumianego jako „jeden interfejs, wiele metod”.

Zanim przejdziemy do dalszego poznawania interfejsów, musimy zwrócić uwagę na jedną ważną informację. Otóż w JKD 8 wprowadzono zmianę w interfejsach, która znacząco wpływa na ich możliwości. We wcześniejszych wersjach języka interfejsy nie mogły definiować żadnych implementacji. A zatem do momentu wprowadzenia JDK 8 interfejsy mogły określać jedynie to, co klasa mogła robić, lecz nie określały, jak to robiła. Jednak w JDK 8 ta sytuacja uległa zmianie. Obecnie istnieje już możliwość dodawania do interfejsu metody **implementacji domyślnej**. Oznacza to, że teraz interfejsy mogą już określać pewne zachowania. Niemniej jednak te domyślne metody stanowią w zasadzie możliwość specjalnego przeznaczenia i bynajmniej nie przekreślają dotychczasowego charakteru i znaczenia interfejsów. Dlatego jako zasadę ogólną należy przyjąć, że tak jak było dotychczas, przeważnie będziemy tworzyć i stosować interfejsy pozbawione metod domyślnych. Dlatego też zacznę od przedstawienia interfejsów w ich tradycyjnej postaci, a metodę domyślną interfejsów opiszę pod koniec rozdziału.

Poniżej przedstawiłem uproszczoną ogólną postać tradycyjnej deklaracji interfejsu:

```
dostęp interface nazwa{
    typ-zwracany nazwa-metody1(lista-parametrów);
    typ-zwracany nazwa-metody2(lista-parametrów);
    typ zmienna1 = wartość;
    typ zmienna2 = wartość;
    //...
    typ-zwracany nazwa-metodyN(lista-parametrów);
    typ zmiennaN = wartość;
}
```

W tym przypadku `dostęp` oznacza modyfikator `public` albo nie jest używany. W tym drugim przypadku zostaje zastosowany domyślny poziom dostępu i interfejs jest dostępny jedynie wewnątrz pakietu. Jeśli zadeklarujemy interfejs jako `public`, może być on używany przez dowolny kod (pamiętaj jednak, że interfejs zadeklarowany jako `public` musi znajdować się w pliku o nazwie takiej samej jak nazwa interfejsu). *Nazwa-interfejsu* może być dowolnym, poprawnym identyfikatorem.

W przypadku interfejsów o tradycyjnej postaci w metodach deklarujemy jedynie typ zwracany i sygnaturę. Metody interfejsu są metodami abstrakcyjnymi. Każda klasa implementująca interfejs musi implementować wszystkie jego metody. Metody interfejsu są niejawnie zadeklarowane jako `public`.

Zmienne interfejsu nie są zmiennymi instancjami. Niejawnie są zadeklarowane jako `public`, `final` i `static` i wymagają inicjalizacji. W efekcie spełniają jedynie rolę stałych. Na listingu 8.7 przedstawiłem przykład interfejsu. Specyfikuje on interfejs klasy generującej sekwencję liczb.

Listing 8.7. *Series.java*

```
public interface Series {
    int getNext(); // zwraca następną element sekwencji
    void reset(); // restart
    void setStart(int x); // określa wartość początkową
}
```

Interfejs ten został zadeklarowany jako `public` i wobec tego może zostać zaimplementowany w dowolnym pakiecie.

Implementacje interfejsów

Po zdefiniowaniu interfejsu mogą go implementować różne klasy. Aby zaimplementować interfejs, umieszczamy w definicji klasy klauzulę `implements`, a następnie tworzymy implementację metod wymaganych przez interfejs. Ogólna postać deklaracji klasy zawierającej klauzulę `implements` przedstawia się następująco:

```
class nazwklasy extends klasabazowa implements interfejs {
    // ciało-klasy
}
```

Jeśli klasa ma stanowić implementację więcej niż jednego interfejsu, to nazwy interfejsów rozdziela się przecinkami. Oczywiście klauzula `extends` jest opcjonalna.

Metody implementujące interfejs muszą zostać zadeklarowane jako `public`. Ich sygnatury muszą dokładnie odpowiadać sygnaturom metod, które zawiera definicja interfejsu.

Na listingu 8.8 przedstawiłem przykład klasy implementującej przedstawiony wcześniej interfejs `Series`. Klasa `ByTws` generuje serię liczb, w której kolejna liczba jest większa o 2 od poprzedniej.

Listing 8.8. `ByTws.java`

```
// Implementuje interfejs Series
class ByTws implements Series {
    int start;
    int val;
    ByTws() {
        start = 0;
        val = 0;
    }
    public int getNext() {
        val += 2;
        return val;
    }
    public void reset() {
        val = start;
    }
    public void setStart(int x) {
        start = x;
        val = x;
    }
}
```

↑ Implementuje interfejs Series.

Zwróć uwagę, że metody `getNext()`, `reset()` i `setStart()` zostały zadeklarowane jako `public`. Jest to obowiązkowe. Gdy implementujesz metodę interfejsu, musisz zadeklarować ją jako `public`, gdyż wszystkie składowe interfejsy są niejawnie zadeklarowane jako `public`.

Na listingu 8.9 przedstawiłem klasę demonstrującą działanie klasy `ByTws`.

Listing 8.9. `SeriesDemo.java`

```
class SeriesDemo {
    public static void main(String args[]) {
        ByTws ob = new ByTws();

        for(int i=0; i < 5; i++)
            System.out.println("Następna wartość: " +
                ob.getNext());

        System.out.println("\nRestart");
        ob.reset();
    }
}
```

```

for(int i=0; i < 5; i++)
    System.out.println("Następna wartość: " +
        ob.getNext());

System.out.println("\nRozpoczynam od 100");
ob.setStart(100);
for(int i=0; i < 5; i++)
    System.out.println("Następna wartość: " +
        ob.getNext());
}
}

```

A oto wynik działania tego programu:

```

Następna wartość: 2
Następna wartość: 4
Następna wartość: 6
Następna wartość: 8
Następna wartość: 10

```

Restart

```

Następna wartość: 2
Następna wartość: 4
Następna wartość: 6
Następna wartość: 8
Następna wartość: 10

```

Rozpoczynam od 100

```

Następna wartość: 102
Następna wartość: 104
Następna wartość: 106
Następna wartość: 108
Następna wartość: 110

```

Klasy implementujące interfejsy mogą również definiować dodatkowe, własne składowe. Na przykład wersja klasy `ByTws` przedstawiona na listingu 8.10 dodaje własną metodę `getPrevious()` zwracającą poprzednią wartość.

Listing 8.10. *ByTws.java*

// Implementuje interfejs Series i dodaje metodę getPrevious().

```

class ByTws implements Series {
    int start;
    int val;
    int prev;
    ByTws() {
        start = 0;
        val = 0;
        prev = -2;
    }

    public int getNext() {
        prev = val;
        val += 2;
        return val;
    }

    public void reset() {
        val = start;
        prev = start-2;
    }

    public void setStart(int x) {
        start = x;
    }
}

```



```

    val = x;
    prev = x - 2;
}

int getPrevious() { ← Dodaje metodę, której nie definiuje interfejs Series.
    return prev;
}
}

```

Zwróć uwagę, że wprowadzenie metody `getPrevious()` wymagało zmian w implementacji metod zdefiniowanych przez interfejs `Series`. Ponieważ jednak interfejs metod pozostał taki sam, zmiany nie mają żadnego wpływu na działanie programu demonstrującego klasę `ByTwos`. Jest to jedna z zalet stosowania interfejsów.

Jak już wspominałem, interfejs może implementować dowolna liczba klas. Na listingu 8.11 przedstawiłem klasę `ByThrees` generującą sekwencję wielokrotności liczby 3.

Listing 8.11. *ByThrees.java*

```

// Implementuje interfejs Series
class ByThrees implements Series { ← Implementuje interfejs Series w inny sposób.
    int start;
    int val;

    ByThrees() {
        start = 0;
        val = 0;
    }

    public int getNext() {
        val += 3;
        return val;
    }

    public void reset() {
        start = 0;
        val = 0;
    }

    public void setStart(int x) {
        start = x;
        val = x;
    }
}

```

I jeszcze jedno: jeśli zadeklarowałeś, że klasa implementuje interfejs, ale nie dostarczyłeś pełnej implementacji wszystkich metod zdefiniowanych przez interfejs, to musisz zadeklarować tę klasę jako `abstract`. Oczywiście nie możesz tworzyć obiektów takiej klasy, ale możesz użyć jej jako abstrakcyjnej klasy bazowej, pozostawiając jej klasom pochodnym dostarczenie kompletnej implementacji.

Referencje interfejsu

Możesz być nieco zaskoczony możliwością zadeklarowania zmiennej, której typem jest interfejs. Innymi słowy, tworzysz w ten sposób zmienną referencyjną interfejsu. Możesz jej użyć, aby odwołać się do dowolnego obiektu, który implementuje ten interfejs. Gdy wywołujesz metodę dla obiektu, do którego odwołujesz się za pomocą referencji interfejsu, wykonana zostanie wersja metody implementowana przez ten konkretny obiekt. Proces ten jest podobny do stosowania referencji klasy bazowej do obiektów klas pochodnych, co omówiłem w rozdziale 7.

Przykład użycia referencji interfejsu przedstawiłem na listingu 8.12. Używa on tej samej zmiennej referencyjnej interfejsu do wywoływania metod obiektów klas `ByTwos` i `ByThrees`.

Listing 8.12. *SeriesDemo2.java**// Demonstruje użycie referencji interfejsu.*

```

class ByTwos implements Series {
    int start;
    int val;

    ByTwos() {
        start = 0;
        val = 0;
    }

    public int getNext() {
        val += 2;
        return val;
    }

    public void reset() {
        val = start;
    }

    public void setStart(int x) {
        start = x;
        val = x;
    }
}

class ByThrees implements Series {
    int start;
    int val;

    ByThrees() {
        start = 0;
        val = 0;
    }

    public int getNext() {
        val += 3;
        return val;
    }

    public void reset() {
        val = start;
    }

    public void setStart(int x) {
        start = x;
        val = x;
    }
}

class SeriesDemo2 {
    public static void main(String args[]) {
        ByTwos twoOb = new ByTwos();
        ByThrees threeOb = new ByThrees();
        Series ob;

        for(int i=0; i < 5; i++) {
            ob = twoOb;
            System.out.println("Kolejna wartość sekwencji ByTwos: " +
                ob.getNext()); ←
            ob = threeOb;
            System.out.println("Kolejna wartość sekwencji ByThrees: " +
                ob.getNext()); ←
        }
    }
}

```

Dostęp do obiektu
za pomocą
referencji interfejsu.

```

    }
}
}

```

W metodzie `main()` zmienna `ob` została zadeklarowana jako zmienna referencyjna, której typem jest interfejs `Series`. Oznacza to, że możesz jej przypisać referencję dowolnego obiektu, który implementuje interfejs `Series`. W tym przypadku zmienna jest używana dla obiektów `twoOb` i `threeOb`, które należą odpowiednio do klas `ByTwos` i `ByThrees`. Obie wymienione klasy implementują interfejs `Series`. Zmienna referencyjna interfejsu zna jedynie metody zdefiniowane przez interfejs. Zatem za pomocą zmiennej `ob` nie uzyskasz dostępu do dodatkowych składowych lub metod, które może obsługiwać konkretny obiekt.

Przykład 8.1. Tworzymy interfejs `Queue`

`ICharQ.java`
`IQDemo.java`

Działanie interfejsów zademonstruję na praktycznym przykładzie. We wcześniejszych rozdziałach stworzyliśmy klasę `Queue` implementującą kolejkę znaków o stałym rozmiarze. Istnieje jednak wiele innych sposobów implementacji kolejek. Kolejki mogą mieć stały rozmiar lub „rosnąć”. Mogą być liniowe, co szybko może doprowadzić do ich przepełnienia, albo cykliczne, co pozwala zawsze umieszczać w nich kolejne elementy, jeśli tylko inne są pobierane z kolejki. Elementy kolejki mogą być przechowywane w tablicach, listach, drzewie binarnym i innych strukturach danych. Niezależnie od sposobu implementacji kolejki jej interfejs pozostaje taki sam. Definiują go metody `put()` i `get()`. Ponieważ możemy oddzielić interfejs kolejki od jej implementacji, to łatwo możemy zdefiniować interfejs kolejki, pozostawiając szczegóły różnym implementacjom.

W tym przykładzie stworzysz interfejs kolejki przechowującej znaki i trzy jego implementacje. Wszystkie trzy implementacje będą przechowywać znaki w tablicy. Pierwsza implementacja będzie liniową kolejką o stałym rozmiarze, którą stworzyliśmy już w poprzednich rozdziałach. Druga implementacja będzie kolejką cykliczną. Gdy implementacja takiej kolejki dotrze do końca tablicy, indeksy operacji `put` i `get` automatycznie powracają na początek tablicy. Dzięki temu do kolejki może trafić dowolnie wiele elementów, pod warunkiem że równocześnie są z niej pobierane inne elementy. Ostatnia z implementacji będzie tworzyć dynamiczną kolejkę, zwiększającą swój rozmiar, gdy to konieczne.

1. Utwórz plik `ICharQ.java` i umieść w nim poniższą definicję interfejsu:

```

// Interfejs kolejki przechowującej znaki.
public interface ICharQ {
    // Umieszcza znak w kolejce.
    void put(char ch);

    // Pobiera znak z kolejki.
    char get();
}

```

Interfejs ten jest bardzo prosty, składa się tylko z dwóch metod. Każda klasa implementująca interfejs `ICharQ` będzie musiała zaimplementować te metody.

2. Utwórz plik o nazwie `IQDemo.java`.
3. Na początek umieść w nim poniższą definicję klasy `FixedQueue`:

```

// Kolejka znaków o stałym rozmiarze.
class FixedQueue implements ICharQ {
    private char q[]; // tablica przechowująca elementy kolejki
    private int putloc, getloc; // indeksy operacji put i get

    // Tworzy pustą kolejkę o podanym rozmiarze.
    public FixedQueue(int size) {
        q = new char[size]; // przydziela pamięć kolejce
        putloc = getloc = 0;
    }
}

```

```

// Umieszcza znak w kolejce.
public void put(char ch) {
    if(putloc==q.length) {
        System.out.println(" -- Kolejka pełna.");
        return;
    }

    q[putloc++] = ch;
}

// Pobiera znak z kolejki.
public char get() {
    if(getloc == putloc) {
        System.out.println(" -- Kolejka pusta.");
        return (char) 0;
    }

    return q[getloc++];
}
}

```

Tę implementację interfejsu ICharQ zaadaptowaliśmy z klasy Queue z rozdziału 5. i jej działanie powinno być dla Ciebie zrozumiałe.

4. Dodaj w pliku *IQDemo.java* klasę *CircularQueue* przedstawioną poniżej. Implementuje ona cykliczną kolejkę znaków.

```

// Kolejka cykliczna.
class CircularQueue implements ICharQ {
    private char q[]; // tablica przechowująca elementy kolejki
    private int putloc, getloc; // indeksy operacji put i get

    // Tworzy pustą kolejkę o podanym rozmiarze.
    public CircularQueue(int size) {
        q = new char[size+1]; // przydziela pamięć kolejce
        putloc = getloc = 0;
    }

    // Umieszcza znak w kolejce.
    public void put(char ch) {
        /* Kolejka jest pełna, jeśli putloc ma wartość mniejszą
        o jeden niż getloc lub gdy putloc wskazuje koniec tablicy,
        a getloc początek. */
        if(putloc+1==getloc |
            ((putloc==q.length-1) & (getloc==0))) {
            System.out.println(" -- Kolejka pełna.");
            return;
        }

        q[putloc++] = ch;
        if(putloc==q.length) putloc = 0; // powrót na początek tablicy
    }

    // Pobiera znak z kolejki.
    public char get() {
        if(getloc == putloc) {
            System.out.println(" -- Kolejka pusta.");
            return (char) 0;
        }
        char ch = q[getloc++];
        if(getloc==q.length) getloc = 0; // powrót na początek tablicy
        return ch;
    }
}

```

Kolejka cykliczna umożliwia ponowne wykorzystanie miejsca w tabeli zwolnionego przez elementy pobrane z kolejki. Dzięki temu w kolejce cyklicznej można umieszczać dowolną liczbę elementów, pod warunkiem że inne elementy są z niej pobierane. Chociaż implementacja działania takiej kolejki jest prosta i sprowadza się do wyzerowania odpowiedniego indeksu, gdy kolejka osiągnie koniec tablicy, to warunki brzegowe są na pierwszy rzut oka nieco bardziej skomplikowane. Kolejka cykliczna nie jest pełna, gdy osiągnie koniec tablicy, ale gdy umieszczenie kolejnego elementu spowodowałoby nadpisanie elementu, który nie został jeszcze pobrany z kolejki. Zatem metoda `put()` musi sprawdzić kilka warunków, aby ustalić, czy kolejka jest pełna. Jak sugerują to komentarze umieszczone w kodzie, kolejka jest pełna, gdy wartość indeksu `putloc` jest o jeden mniejsza niż indeksu `getloc`, albo gdy `putloc` znajduje się na końcu tablicy, a `getloc` na początku. Kolejka jest pusta, gdy wartości indeksów `getloc` i `putloc` są równe. Aby ułatwić sprawdzanie tych warunków, tablica, w której jest przechowywana kolejka, jest o jeden większa od długości kolejki.

- Umieść w pliku `IQDemo.java` klasę `DynQueue` przedstawioną poniżej. Implementuje ona kolejkę zwiększającą swój rozmiar, gdy kolejka zapełni się.

```
// Kolejka dynamiczna.
class DynQueue implements ICharQ {
    private char q[]; // tablica przechowująca elementy kolejki
    private int putloc, getloc; // indeksy operacji put i get

    // Tworzy pustą kolejkę o podanym rozmiarze.
    public DynQueue(int size) {
        q = new char[size]; // przydziela pamięć kolejce
        putloc = getloc = 0;
    }

    // Umieszcza znak w kolejce.
    public void put(char ch) {
        if(putloc==q.length) {
            // zwiększa rozmiar kolejki
            char t[] = new char[q.length * 2];

            // kopiuje elementy do nowej kolejki
            for(int i=0; i < q.length; i++)
                t[i] = q[i];

            q = t;
        }
        q[putloc++] = ch;
    }

    // Pobiera znak z kolejki.
    public char get() {
        if(getloc == putloc) {
            System.out.println(" -- Kolejka pusta.");
            return (char) 0;
        }
        return q[getloc++];
    }
}
```

W tej implementacji próba umieszczenia znaku w pełnej kolejce powoduje utworzenie dwa razy większej tablicy, do której zostaje skopiowana zawartość kolejki. Referencja nowej tablicy zostaje przypisana składowej `q`.

- Aby zademonstrować działanie trzech implementacji interfejsu `ICharQ`, umieść w pliku `IQDemo.java` poniższą definicję klasy. Używa ona referencji interfejsu `ICharQ`, odwołując się do wszystkich trzech kolejek.

```

// Demonstruje interfejs ICharQ.
class IQDemo {
    public static void main(String args[]) {
        FixedQueue q1 = new FixedQueue(10);
        DynQueue q2 = new DynQueue(5);
        CircularQueue q3 = new CircularQueue(10);

        ICharQ iQ;

        char ch;
        int i;

        iQ = q1;
        // Umieszcza znaki w kolejce o stałym rozmiarze.
        for(i=0; i < 10; i++)
            iQ.put((char) ('A' + i));

        // Wyświetla zawartość kolejki.
        System.out.print("Zawartość kolejki o stałym rozmiarze: ");
        for(i=0; i < 10; i++) {
            ch = iQ.get();
            System.out.print(ch);
        }
        System.out.println();

        iQ = q2;
        // Umieszcza znaki w kolejce dynamicznej.
        for(i=0; i < 10; i++)
            iQ.put((char) ('Z' - i));

        // Wyświetla zawartość kolejki.
        System.out.print("Zawartość kolejki dynamicznej: ");
        for(i=0; i < 10; i++) {
            ch = iQ.get();
            System.out.print(ch);
        }

        System.out.println();

        iQ = q3;
        // Umieszcza znaki w kolejce cyklicznej.
        for(i=0; i < 10; i++)
            iQ.put((char) ('A' + i));
        // Wyświetla zawartość kolejki.
        System.out.print("Zawartość kolejki cyklicznej: ");
        for(i=0; i < 10; i++) {
            ch = iQ.get();
            System.out.print(ch);
        }

        System.out.println();

        // Umieszcza więcej znaków w kolejce cyklicznej.
        for(i=10; i < 20; i++)
            iQ.put((char) ('A' + i));

        // Wyświetla zawartość kolejki.
        System.out.print("Zawartość kolejki cyklicznej: ");
        for(i=0; i < 10; i++) {
            ch = iQ.get();
            System.out.print(ch);
        }
    }
}

```

```

System.out.println("\nUmieszczam i pobieram znak" +
    " z kolejki cyklicznej.");

// Umieszcza i pobiera znak.
for(i=0; i < 20; i++) {
    iQ.put((char) ('A' + i));
    ch = iQ.get();
    System.out.print(ch);
}
}
}

```

7. Poniżej przedstawiłem wynik działania tego programu:

```

Zawartość kolejki o stałym rozmiarze: ABCDEFGHIJ
Zawartość kolejki dynamicznej: ZYXWVUTSRQ
Zawartość kolejki cyklicznej: ABCDEFGHIJ
Zawartość kolejki cyklicznej: KLMNOPQRST
Umieszczam i pobieram znak z kolejki cyklicznej.
ABCDEFGHIJKLMNOPQRST

```

8. A oto kilka zadań dla Ciebie. Utwórz cykliczną wersję implementacji `DynQueue`. Rozszerz interfejs `ICharQ` o metodę `reset()` przywracającą stan wyjściowy kolejki. Utwórz metodę `static` kopiującą zawartość kolejki jednego typu do kolejki innego typu.

Zmienne w interfejsach

Jak już wspomniałem, interfejs może deklarować zmienne, ale niejawnie przyjmuje się, że zmienne te są zadeklarowane jako `public`, `static` i `final`. Może więc wydawać Ci się, że ich przydatność jest bardzo ograniczona, ale nie jest to prawdą. Skomplikowane programy wykorzystują wiele stałych opisujących rozmiary tablic, różne ograniczenia, wartości specjalne i tym podobne. Ponieważ kod takich programów zwykle znajduje się w wielu plikach źródłowych, to musi istnieć wygodny sposób udostępniania stałych w każdym pliku. W języku Java można użyć w tym celu interfejsu.

Aby zdefiniować zbiór stałych współdzielonych przez różne pliki źródłowe, utwórz interfejs, który zawiera tylko te stałe i nie definiuje żadnych metod. Każda klasa, która używa tych stałych, będzie „implementować” ten interfejs, dzięki czemu uzyska do nich dostęp. Przykład takiego zastosowania interfejsu przedstawiłem na listingu 8.13.

Listing 8.13. `IConstD.java`

```

// Interfejs zawierający stałe.
interface IConst {
    int MIN = 0;
    int MAX = 10;
    String ERRORMSG = "Błąd zakresu";
}

class IConstD implements IConst {
    public static void main(String args[]) {
        int nums[] = new int[MAX];

        for(int i=MIN; i < 11; i++) {
            if(i >= MAX) System.out.println(ERRORMSG);
            else {
                nums[i] = i;
                System.out.print(nums[i] + " ");
            }
        }
    }
}

```

} — To są stałe.



Uwaga

Technika stosowania interfejsów w celu definiowania często używanych stałych jest nieco kontrowersyjna. Opisałem ją tu wyłącznie w celu wyczerpującej prezentacji możliwości i zastosowań interfejsów.

Interfejsy mogą dziedziczyć

Interfejs może dziedziczyć po innym interfejsie poprzez zastosowanie słowa kluczowego `extends`. Składnia jest taka sama jak w przypadku dziedziczenia klas. Jeśli klasa implementuje interfejs, który dziedziczy po innym interfejsie, musi dostarczyć implementacji dla wszystkich metod wymaganych przez łańcuch dziedziczenia interfejsów. Ilustruje to przykład przedstawiony na listingu 8.14.

Listing 8.14. *IFExtend.java*

```
// Dziedziczenie interfejsów.
interface A {
    void meth1();
    void meth2();
}

// B dziedziczy meth1() i meth2() i dodaje meth3().
interface B extends A { ← B dziedziczy po A.
    void meth3();
}

// Ta klasa musi implementować wszystkie metody interfejsów A i B
class MyClass implements B {
    public void meth1() {
        System.out.println("Implementacja meth1().");
    }

    public void meth2() {
        System.out.println("Implementacja meth2().");
    }

    public void meth3() {
        System.out.println("Implementacja meth3().");
    }
}

class IFExtend {
    public static void main(String arg[]) {
        MyClass ob = new MyClass();

        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}
```

W ramach eksperymentu możesz spróbować usunąć implementację metody `meth1()` z klasy `MyClass`. Przekonasz się, że spowoduje to błąd kompilacji. Jak wspomniałem wcześniej, każda klasa implementująca interfejs musi implementować wszystkie metody tego interfejsu, włączając w to metody dziedziczone po innych interfejsach.

Domyślne metody interfejsów

Jak już pisałem wcześniej, przed udostępnieniem JDK 8 interfejsy nie mogły zawierać żadnych implementacji. Oznacza to, że we wszystkich wcześniejszych wersjach języka Java wszystkie metody interfejsów musiały być metodami abstrakcyjnymi, czyli pozbawionymi ciała. To tradycyjna forma interfejsów,

której dotyczyły wszystkie informacje podane we wcześniejszej części rozdziału. Jednak sytuacja zmieniła się po wprowadzeniu JDK 8, gdyż w tej wersji język Java został rozszerzony o możliwość dodawania do interfejsów tak zwanej **metody domyślnej**. Metoda domyślna pozwala na zdefiniowanie domyślnej implementacji metody interfejsu. Innymi słowy, dzięki wprowadzeniu metody domyślnej metoda interfejsu może już posiadać ciało — nie musi być metodą abstrakcyjną. W trakcie prac nad wprowadzeniem tej możliwości metody domyślne nazywano także **metodami rozszerzeń**, dlatego obecnie można się spotkać z oboma tymi terminami.

Głównym powodem wprowadzenia metod domyślnych była chęć zapewnienia sposobu rozszerzania interfejsu bez wywoływania problemów w już istniejącym kodzie. Pamiętajsz zapewne, że wszystkie metody zdefiniowane w interfejsie muszą zostać zaimplementowane. W przeszłości, kiedy do popularnych i często używanych interfejsów dodawano nowe metody, mogło to doprowadzić do problemów w już istniejącym kodzie, spowodowanych brakiem implementacji tych nowych metod. Wprowadzenie metody domyślnej rozwiązuje ten problem, gdyż zapewnia możliwość dostarczenia implementacji, która będzie używana, jeśli nie zostanie jawnie podana żadna inna implementacja. A zatem dodanie metody domyślnej nie wywoła żadnych problemów w już istniejącym kodzie.

Kolejnym powodem wprowadzenia metod domyślnych była chęć zapewnienia możliwości określania w interfejsach metod, które zależnie do sposobu używania danego interfejsu byłyby w zasadzie opcjonalne. Na przykład interfejs może definiować grupę metod służących do operowania na sekwencji elementów. Jedna z nich może nosić nazwę `remove()` i służyć do usuwania elementu z sekwencji. Niemniej jednak jeśli ten sam interfejs ma służyć do obsługi zarówno sekwencji o zmiennej zawartości, jak i takich, których zawartość nie może ulegać zmianie, to metoda `remove()` właściwie będzie metodą opcjonalną, gdyż sekwencje o niezmienniej zawartości nie będą z niej korzystać. W przeszłości klasa stanowiąca implementację sekwencji o niezmienniej zawartości musiała definiować pustą implementację metody `remove()`, niezależnie od tego, że nigdy by jej nie używała. Obecnie w takim interfejsie można by umieścić domyślną implementację metody `remove()`, która niczego nie zrobi lub zgłosi błąd. Udostępnienie tej metody domyślnej sprawi, że klasa reprezentująca sekwencję o niezmienniej zawartości nie będzie już musiała definiować swojej własnej, pustej wersji metody `remove()`. A zatem dostarczenie metody domyślnej sprawia, że implementacja metody `remove()` staje się czynnością opcjonalną, a nie wymaganą.

Koniecznym należy zaznaczyć, że wprowadzenie metod domyślnych w żaden sposób nie zmienia kluczowego aspektu interfejsów: wciąż nie można w nich definiować zmiennych instancyjnych. Czyli kluczowa różnica pomiędzy interfejsami i klasami polega na tym, że klasy mogą zachowywać stan, a interfejsy nie. Co więcej, wciąż nie można utworzyć instancji interfejsu, dysponując wyłącznie nim samym. Konieczna jest do tego klasa, która będzie go implementować. A zatem mimo że zaczynając od JDK 8, interfejsy mogą definiować metody domyślne, to jednak aby utworzyć instancję interfejsu, musi on zostać zaimplementowany w jakiejś klasie.

I ostatnia sprawa: jako ogólną zasadę należy przyjąć fakt, że metody domyślne stanowią rozwiązanie specjalnego przeznaczenia. Interfejsy, które będziemy tworzyć w przyszłości, wciąż będą przede wszystkim określać, co należy zrobić, a nie jak te operacje wykonywać. Niemniej jednak dodanie metod domyślnych zapewni nam dodatkową elastyczność.

Podstawowe informacje o metodach domyślnych

Metoda domyślna interfejsu jest definiowana bardzo podobnie do zwyczajnych metod w klasach. Jedyna różnica polega na tym, że deklaracja metody domyślnej jest poprzedzana słowem kluczowym `default`. W ramach przykładu przeanalizuj poniższy prosty interfejs (listing 8.15):

Listing 8.15. *MyIF.java*

```
public interface MyIF {
    // To jest "normalna" deklaracja metody interfejsu.
    // NIE definiuje ona domyślnej implementacji.
    int getUserID();
}
```

```
// A to jest metoda domyślna. Zauważ, że podaje ona
// domyślną implementację.
default int getAdminID() {
    return 1;
}
}
```

Interfejs `MyIF` deklaruje dwie metody. Pierwsza z nich, `getUserID()`, jest standardową deklaracją metody interfejsu. Nie definiuje ona żadnej implementacji. Z kolei druga metoda, `getAdminID()`, jest metodą domyślną i zawiera domyślną implementację. W powyższym przykładzie metoda domyślna jedynie zwraca wartość 1. Zwróć szczególną uwagę na postać deklaracji metody `getAdminID()`. Została ona poprzedzona słowem kluczowym `default`. Tę składnię można opisać bardziej ogólnie: w celu zdefiniowania metody domyślnej jej deklarację należy poprzedzić słowem kluczowym `default`.

Ponieważ metoda `getAdminID()` zawiera domyślną implementację, zatem nie trzeba jej przesłaniać w klasie implementującej interfejs `MyIF`. Innymi słowy, jeśli klasa implementująca interfejs nie poda własnej implementacji tej metody, to zostanie użyta implementacja domyślna. Na przykład prosta klasa `MyIFImp` przedstawiona na listingu 8.16 jest całkowicie prawidłowa:

Listing 8.16. `MyIFImp.java`

```
// Klasa implementująca interfejs MyIF.
class MyIFImp implements MyIF {
    // Tylko metoda getUserID() zdefiniowana w interfejsie MyIF musi zostać
    // zaimplementowana. W przypadku metody getAdminID() możemy
    // skorzystać z jej domyślnej implementacji.
    public int getUserID() {
        return 100;
    }
}
```

Przykład przedstawiony na listingu 8.17 tworzy instancję klasy `MyIFImp` i używa jej w celu wywołania metod `getUserID()` i `getAdminID()`.

Listing 8.17. `DefaultMethodDemo.java`

```
// Zastosowanie metody domyślnej.
class DefaultMethodDemo {
    public static void main(String args[]) {

        MyIFImp obj = new MyIFImp();

        // Metodę getUserID() możemy wywołać, gdyż została jawnie
        // zaimplementowana w klasie MyIFImp:
        System.out.println("Identyfikator użytkownika to: " + obj.getUserID());

        // Możemy także wywołać metodę getAdminID(), gdyż jest ona
        // metodą domyślną i także posiada implementację:
        System.out.println("Identyfikator administratora to: " + obj.getAdminID());
    }
}
```

Powyższy program zwróci następujące wyniki:

```
Identyfikator użytkownika to 100
Identyfikator administratora to 1
```

Jak widać, automatycznie została użyta domyślna implementacja metody `getAdminID()`. Definiowanie jej w klasie `MyIFImp` nie było konieczne. A zatem w przypadku metody `getAdminID()` implementowanie jej w klasie jest opcjonalne. (Oczywiście gdyby klasa wymagała zwrócenia innej wartości identyfikatora administratora, to zaimplementowanie metody `getAdminID()` byłoby konieczne).

Klasa implementująca interfejs oczywiście może podać własną implementację jego metody domyślnej, co więcej, takie rozwiązanie jest bardzo często stosowane. Na przykład klasa `MyIFImp2` przedstawiona na listingu 8.18 przesłania metodę `getAdminID()`:

Listing 8.18. *MyIFImp2.java*

```
class MyIFImp2 implements MyIF {
    // W tej wersji klasy podaliśmy implementacje obu metod:
    // getUserID() oraz getAdminID().
    public int getUserID() {
        return 100;
    }

    public int getAdminID() {
        return 42;
    }
}
```

W tym przypadku wywołanie metody `getAdminID()` zwróci wartość inną niż domyślna.

Praktyczny przykład metody domyślnej

Chociaż poprzednie przykłady zaprezentowały mechanikę stosowania metody domyślnej, to jednak nie wykazały jej przydatności w zastosowaniach praktycznych. W tym celu wrócimy do interfejsu `Series` przedstawionego we wcześniejszej części rozdziału. Na potrzeby rozważań przyjmijmy, że interfejs ten jest bardzo popularny i korzysta z niego bardzo wielu programistów. Co więcej, założmy, że na podstawie analizy wzorców zastosowania udało się ustalić, że do bardzo wielu klas implementujących ten interfejs dodawana była metoda zwracająca tablicę zawierającą n kolejnych elementów serii. Z tego względu podjąłeś decyzję o rozszerzeniu interfejsu i dodaniu do niego takiej metody; ma to być metoda `getNextArray()` o następującej postaci:

```
int[] getNextArray(int n)
```

Gdzie n określa liczbę pobieranych elementów serii. Przed wprowadzeniem metod domyślnych taka modyfikacja interfejsu `Series` spowodowałaby awarię istniejącej bazy kodu, gdyż jego istniejące implementacje nie definiowałyby metody `getNextArray()`. Niemniej jednak dzięki podaniu domyślnej implementacji tej nowej metody można ją dodać do interfejsu bez wywoływania żadnych problemów.

W niektórych przypadkach, kiedy do istniejącego interfejsu zostanie dodana metoda domyślna, w razie próby jej użycia klasa, która go implementuje, zgłosi błąd. Takie rozwiązanie jest konieczne, gdy nie można znaleźć implementacji metod domyślnych, których można użyć we wszystkich możliwych sytuacjach. Takie typy metod domyślnych definiują kod, który właściwie można uznać za opcjonalny. Jednak w niektórych przypadkach można zdefiniować metodę domyślną, która będzie działać we wszystkich sytuacjach. Dotyczy to także naszej przykładowej metody `getNextArray()`. Ponieważ interfejs `Series` wymaga implementacji metody `getNext()`, więc domyślna implementacja metody `getNextArray()` może z niej skorzystać. A zatem poniższy listing 8.19 przedstawia interfejs `Series`, stanowiący nową wersję interfejsu `Series` i zawierający metodę domyślną o nazwie `getNextArray()`:

Listing 8.19. *Series2.java*

```
// Rozbudowana wersja interfejsu Series, zawierająca
// metodę domyślną o nazwie getNextArray().
public interface Series {
    int getNext(); // Zwraca następną liczbę w serii.

    // Zwraca tablicę zawierającą n następnych elementów
    // serii.
    default int[] getNextArray(int n) {
```

```

int[] vals = new int[n];

for(int i=0; i < n; i++) vals[i] = getNext();
return vals;
}

void reset(); // restart
void setStart(int x); // określa wartość początkową
}

```

Zwróć szczególną uwagę na sposób, w jaki została zaimplementowana metoda domyślna `getNextArray()`. Ponieważ metoda `getNext()` należała do początkowej specyfikacji interfejsu `Series`, zatem będzie ona dostępna w każdej klasie, która go implementuje. Dlatego można jej użyć w metodzie `getNextArray()`, by pobrać n następných elementów serii. W efekcie każda klasa implementująca rozszerzoną wersję interfejsu `Series` będzie mogła używać metody `getNextArray()` bez konieczności jej przesłaniania czy wprowadzania jakichkolwiek dodatkowych modyfikacji. Oznacza to także, że ta zmiana interfejsu `Series` nie doprowadzi do problemów w żadnym już istniejącym kodzie. Oczywiście nic nie stoi na przeszkodzie, by klasa implementująca ten interfejs dostarczyła własną implementację metody `getNextArray()`.

Jak pokazał powyższy przykład, metody domyślne zapewniają dwie podstawowe korzyści:

- Dają możliwość łagodnego, stopniowego przekształcania interfejsu bez wywoływania problemów w istniejącym kodzie.
- Pozwalają dostarczać opcjonalne możliwości funkcjonalne bez konieczności pisania w klasach implementujących pustych metod zastępczych, gdy funkcjonalności te nie są potrzebne.

W przypadku przedstawionej powyżej metody `getNextArray()` szczególnie znaczenie ma ta druga zaleta. Jeśli klasa implementująca interfejs `Series` nie wymaga możliwości, jakie daje metoda `getNextArray()`, to nie będzie musiała zawierać jej pustej implementacji. A to oznacza, że kod tej klasy będzie bardziej przejrzysty.

Problemy wielokrotnego dziedziczenia

Zgodnie z informacjami podanymi we wcześniejszej części książki Java nie udostępnia wielokrotnego dziedziczenia klas. Jednak teraz, kiedy interfejsy mogą zawierać metody domyślne, mógłbyś się zastanawiać, czy nie pozwalają one ominąć tego ograniczenia. Otóż w gruncie rzeczy nie — nie pozwalają. Przypomnij sobie, że wciąż istnieje jedna, podstawowa różnica pomiędzy interfejsami i klasami: klasy mogą przechowywać stan (dzięki zmiennym instancyjnym), a interfejsy nie.

Pomimo to metody domyślne częściowo oferują możliwości, które normalnie można by powiązać z pojęciem wielokrotnego dziedziczenia. Na przykład można stworzyć klasę implementującą dwa interfejsy. Jeśli każdy z nich będzie definiował metody domyślne, to klasa odziedziczy po nich jakieś zachowania. A zatem metody domyślne w pewnym stopniu faktycznie udostępniają możliwość wielokrotnego dziedziczenia zachowań. Jak się można spodziewać, w takim przypadku pojawia się także możliwość występowania konfliktów nazw.

W ramach przykładu założmy, że klasa `MyClass` implementuje dwa interfejsy: `Alpha` i `Beta`. Co się stanie, jeśli w obu tych interfejsach będzie dostępna metoda `reset()` posiadająca domyślną implementację? Czy klasa `MyClass` użyje wersji metody pochodzącej z interfejsu `Alpha`, czy z interfejsu `Beta`? A co by się stało w sytuacji, gdyby interfejs `Beta` rozszerzał interfejs `Alpha`? Która z wersji metody domyślnej zostałaby użyta? A co by się stało, gdyby klasa `MyClass` udostępniała własną implementację metody `reset()`? Aby rozwiązywać zarówno te, jak i inne podobne problematyczne sytuacje, Java udostępni zestaw reguł pozwalających na likwidowanie takich konfliktów.

Przede wszystkim w każdym przypadku metoda zaimplementowana w klasie będzie mieć pierwszeństwo przed metodami domyślnymi zdefiniowanymi w interfejsach. A zatem jeśli klasa `MyClass` udostępni własną wersję metody `reset()`, przesłaniającą metody domyślne zdefiniowane w interfejsach, to właśnie ona będzie używana. Stanie się tak, nawet jeśli klasa `MyClass` będzie implementować oba interfejsy — `Alpha` i `Beta`. W takim przypadku obie implementacje metod domyślnych zostaną przesłonięte przez metodę klasy `MyClass`.

Dodatkowo zostanie zgłoszony błąd, jeśli klasa implementuje dwa interfejsy, z których każdy udostępni tę samą metodę domyślną, oraz jeśli metoda ta nie została zaimplementowana w samej klasie. Wracając do naszego przykładu: jeśli klasa `MyClass` będzie implementować interfejsy `Alpha` i `Beta`, lecz nie udostępni implementacji metody `reset()`, to zostanie zgłoszony błąd.

W przypadkach, gdy jeden interfejs dziedziczy po drugim i oba udostępniają tę samą metodę domyślną, pierwszeństwo będzie miała wersja metody zdefiniowana w interfejsie dziedziczącym. A zatem — wracając do naszego przykładu — gdyby interfejs `Beta` rozszerzał interfejs `Alpha`, to zostałaby użyta metoda `reset()` zdefiniowana w interfejsie `Beta`.

Do metody domyślnej można odwołać się jawnie, używając nowej wersji słowa kluczowego `super`. Poniżej przedstawiłem jego ogólną postać:

```
NazwaInterfejsu.super.nazwaMetody()
```

Na przykład aby w interfejsie `Beta` odwołać się do domyślnej implementacji metody `reset()` zdefiniowanej w interfejsie `Alpha`, należałoby użyć następującej instrukcji:

```
Alpha.super.reset();
```

Stosowanie metod statycznych w interfejsach

W JDK 8 interfejsy udostępniają jeszcze jedną, nową możliwość: definiowanie jednej lub kilku metod **statycznych**. Podobnie jak metody statyczne definiowane w klasach, także metody statyczne definiowane w interfejsach można wywoływać niezależnie od jakichkolwiek obiektów. Oznacza to, że możliwość wywoływania takich metod nie wymaga ani zaimplementowania interfejsu, ani utworzenia jakichkolwiek jego instancji. Zamiast tego takie metody można wywoływać, podając nazwę interfejsu, kropkę oraz nazwę metody. Poniżej przedstawiłem ogólną postać takiego wywołania:

```
NazwaInterfejsu.nazwaMetodyStatycznej
```

Warto zwrócić uwagę, że odpowiada to sposobowi wywoływania metod statycznych zdefiniowanych w klasach.

Poniższy przykład (listing 8.20) przedstawia interfejs `MyIF2`, zmodyfikowaną wersję interfejsu `MyIF`, do którego dodano metodę statyczną o nazwie `getUniversalID()`. W naszym przykładzie metoda ta zwraca wartość 0.

Listing 8.20. `MyIF2.java`

```
public interface MyIF {
    // To jest "normalna" deklaracja metody interfejsu.
    // NIE definiuje ona domyślnej implementacji.
    int getUserID();

    // A to jest metoda domyślna. Zauważ, że podaje ona
    // domyślną implementację.
    default int getAdminID() {
        return 1;
    }

    // To jest metoda statyczna.
    static int getUniversalID() {
        return 0;
    }
}
```

Metodę `getUniversalID()` można wywoływać w następujący sposób:

```
int uID = MyIF2.getUniversalID();
```

Ponieważ metoda `getUniversalID()` jest metodą statyczną, więc jak już wspominałem, możliwość korzystania z niej nie wymaga wcześniejszego implementowania interfejsu `MyIF2` ani tworzenia jego instancji.

I jeszcze ostatnia uwaga: statyczne metody zdefiniowane w interfejsach nie są dziedziczone, i to ani przez klasy implementujące dany interfejs, ani przez interfejsy, które go rozszerzają.

Ostatnie uwagi o pakietach i interfejsach

Chociaż przykłady w tej książce nie będą często używać pakietów ani interfejsów, oba te narzędzia odgrywają istotną rolę w programowaniu w języku Java. Praktycznie każda prawdziwa aplikacja w języku Java korzysta z pakietów, a znaczna część wykorzystuje również interfejsy. Dlatego ważne jest, żebyś dobrze je opanował.

Test sprawdzający

1. Wykorzystując kod z podrozdziału „Przykład 8.1. Tworzymy interfejs Queue”, umieść interfejs `ICharQ` i jego trzy implementacje w pakiecie o nazwie `qpack`. Klasę demonstrującą działanie kolejki, `IQDemo`, pozostaw w pakiecie domyślnym i pokaż, w jaki sposób może ona importować klasy pakietu `qpack` i ich używać.
2. Czym jest przestrzeń nazw? Dlaczego ważna jest możliwość podziału przestrzeni nazw w Javie?
3. Pakiety przechowywane są w _____.
4. Wyjaśnij różnicę pomiędzy modyfikatorem dostępu `protected` a domyślnym poziomem dostępu.
5. Omów oba sposoby użycia klas pakietu przez inne pakiety.
6. „Jeden interfejs, wiele metod” to podstawowa zasada polimorfizmu w Javie. Który z mechanizmów języka stanowi najlepszy przykład jej zastosowania?
7. Ile klas może implementować interfejs? Ile interfejsów może implementować klasa?
8. Czy interfejsy mogą dziedziczyć?
9. Utwórz interfejs klasy `Vehicle` z rozdziału 7. Nazwij go `IVehicle`.
10. Zmienne interfejsu są niejawnie zadeklarowane jako `static` i `final`. Czy można je udostępnić innym częściom programu?
11. Pakiet jest w zasadzie kontenerem klas. Prawda czy fałsz?
12. Który ze standardowych pakietów Javy jest automatycznie importowany do każdego programu?
13. Jakiego słowa kluczowego używa się podczas deklarowania metod domyślnych?
14. Czy zaczynając od JDK 8, można definiować metody statyczne w interfejsach?
15. Załóżmy, że interfejs `ICharQ` przedstawiony w przykładzie 8.1 stał się bardzo popularny i już od kilku lat jest powszechnie używany. Chciałbyś jednak do niego dodać nową metodę, `reset()`, która ma służyć do przywrócenia początkowego, pustego stanu kolejki. W jaki sposób mógłbyś to zrobić bez wywoływania problemów w już istniejącym kodzie, zakładając, że używasz JDK 8?
16. W jaki sposób są wywoływane statyczne metody interfejsów?

Skorowidz

A

Abstract Window Toolkit, *Patrz:* pakiet AWT
adnotacja, 358, 359
 @Deprecated, 359, 360
 @Documented, 359, 360
 @FunctionalInterface, 359, 360
 @Inherited, 359, 360
 @Native, 359
 @Override, 359, 360
 @Repeatable, 359
 @Retention, 359, 360
 @SafeVarargs, 359, 360
 @SuppressWarnings, 360
 @SuppressWarnings, 359
 @Target, 359, 360
 powtarzalna, 359
 znacznikowa, 359, 360
adres URL, 433, 554
algorytm sortowania, *Patrz:* sortowanie
API współbieżności, *Patrz:* współbieżność API
applet, 98, 421, 422, 423, 424, 425, 433, 436, 450, 471
 działanie, 426
 lokalny, 423
 MySwingApplet, 472
 namiastka, 434
 parametr, 431
 podpis, 423
 uruchamianie, 423
architektura
 model-delegat, *Patrz:* model-delegat
 model-widok-kontroler,
 Patrz: model-widok-kontroler
 z wydzielonym modelem, *Patrz:* model-delegat
argument, 106
 niedozwolony, 269
 przekazywany
 przez referencję, 165, 166, 167
 przez wartość, 165
 w wierszu wywołania, 146, 147

wieloznacznym, 372
 ograniczanie, 374, 375
 wyrażenie lambda, 403, 405
 zmienna liczba, 186, 187, 189, 190
ASCII, 47
asercja, 441

B

baner, 427
bezczyńność, 310
biblioteka, 45
 Collections Framework, 482
 Java API, *Patrz:* Java API
 Swing, 422, *Patrz:* Swing
bit znaku, 152
blok
 finally, 265, 282, 283, *Patrz też:* słowo kluczowe
 finally
 obsługi wyjątku, 253
 static, 181
 synchronized, 327
 try, 254, 255, 256, 282, 286, 287, 288
 postać rozszerzona, 267
 zagnieżdżanie, 261
błąd niejednoznaczności, 388
branch mode, *Patrz:* węzeł gałęzi
bufor, 68, 300
buforowanie wierszami, 68

C

closure, *Patrz:* domknięcie

D

dane
 binarne, 288
 sortowanie, 126
 typ, *Patrz:* typ
dekoder, 300

destruktor, 117
dokumentacja, 553
domknięcie, 394
drzewo, 477
dziedziczenie, 160, 193, 206, 345
 hierarchia, 195
 interfejs, 246
 wielobazowe, 195
 wielokrotne, 250

E

elementarny, *Patrz:* typ prosty
etykieta, 87, 88, 481

F

Framework, 138
funkcja matematyczna, 45

G

graf sceny, 477, 481

H

hierarchia
 klas, *Patrz:* klasa hierarchia
 zawierania, 447

I

instrukcja, *Patrz też:* słowo kluczowe
 break, 67, 73, 79, 86, 87, 90
 rozszerzona, 87
 z etykietą, 87, 90
 case, 72
 continue, 67, 91
 default, 72
 goto, 87, 90
 if, 67, 68, 69, 71
 if-else, 155
 if-else-if, 70, 76, 145
 import, 234, 422
 import static, 357, 358
 iteracji, 67
 package, 228
 pętli, *Patrz:* pętla
 pusta, 79
 return, 67, 104, 105, 401, 407
 skoku, 67
 super, 199, 200, 202, 208

switch, 67, 71, 72, 76, 87, 340
 sterowanie, 145, 146
 zagnieżdżanie, 74
synchronizacja, 327
try, 255, 286, 287, 288
wyboru, 67, 340
interfejs, 227, 235, 252
 ActionListener, 437, 454, 457, 470
 AdjustmentListener, 437
 adnotacja, 359
 AutoCloseable, 286
 autor, 552
 BinaryOperator, 417
 ChangeListener, 492
 Cloneable, 269
 Closeable, 286
 Comparable, 377
 ComponentListener, 437
 Consumer, 417
 ContainerListener, 437
 DataInput, 289, 291, 292
 DataOutput, 288, 291, 292
 FocusListener, 437
 Function, 417
 funkcyjny, 394, 395, 396, 398, 417, 471
 sparametryzowany, 401
 implementacja, 236, 237
 domyślna, 236, 246, 247, 250
 ItemListener, 437, 460
 KeyListener, 437
 klauzula
 extends, 237
 implements, 237
 kolejki, 241
 LayoutManager, 449
 List, 482
 ListSelectionListener, 462
 ListSelectionMode, 463
 metoda, *Patrz:* metoda interfejsu
 MouseListener, 437, 440
 MouseMotionListener, 437, 440
 MouseWheelListener, 437
 obiektów nasłuchujących, 436
 ObservableList, 482
 Predicate, 417
 referencja, 239
 Runnable, 310, 311, 318, 394, 428, 453
 Serializable, 554
 składowa statyczna, 356, 358
 sparametryzowany, 378, 380, 401
 stała, 245

Supplier, 417
 TextListener, 437
 Throwable, 288
 UnaryOperator, 417
 użytkownika
 delegat, 447
 graficzny, 275, 422, 424, 445, 446, 475
 WindowListener, 437
 zmienna, *Patrz:* zmienna interfejsu
 wyjątek, 312

J

Java API, 235, 418
 Java Platform Standard Edition 6, 16
 Java SE, 16
 Java SE 8, 17
 Java wersja, 15
 produktu, 16
 programisty, *Patrz:* Java wersja wewnętrzna
 wewnętrzna, 16
 javadoc, 551, 555
 JavaFX, 446, 475, 477, 498
 aplikacja
 struktura, 478
 uruchamianie, 478, 481
 JavaFX Script, 475

K

kanał, 300
 klasa, 52, 97, 109
 AbstractButton, 454, 459
 abstrakcyjna, 220, 223, 269
 ActionEvent, 484
 Applet, 422, 432, 434, 440, 471
 Application, 477, 479
 ArithmeticException, 257
 autor, 552
 bazowa, 193, 195, 206, 345
 konstruktor, 198, 199, 200
 wyjątek, 259, 260
 BorderPane, 477
 BufferedReader, 293, 294, 296
 Button, 484
 ButtonBase, 484, 487
 CheckBox, 487
 Component, 425, 427, 432, 448
 Console, 293
 Container, 432, 448
 Control, 477, 482, 484
 DataInputStream, 288, 289

DataOutputStream, 288, 289
 definiowanie, 98
 Effect, 498
 Error, 254, 266
 ErrorMsg, 167
 Event, 484
 EventObject, 435
 Exception, 254, 270
 FileInputStream, 281
 FileNotFoundException, 281
 FileOutputStream, 281
 FileReader, 298, 299
 FileWriter, 298
 FlowPane, 477, 480
 FXCollections, 492
 Graphics, 422
 GridPane, 477
 Group, 477
 Help, 302
 hierarchia, 275, 276
 wielopoziomowa, 206
 InputStream, 276, 278
 InputStreamReader, 294, 299
 instancja, 99
 IOException, 281
 ItemEvent, 460
 JApplet, 448, 471
 java.lang.Enum, 345
 JButton, 448, 454
 JCheckBox, 448, 459
 JComponent, 448
 JDialog, 448
 JFrame, 448, 450
 JLabel, 448, 450
 JList, 448, 462, 463
 JTextField, 448, 457, 459
 JToggleButton, 459
 JWindow, 448
 kolekcja, 227
 Label, 482
 Labeled, 482, 484
 ListView, 491, 492
 Math, 45, 46
 MouseEvent, 437
 MouseEvents, 440
 MultipleSelectionModel, 492
 MyThread, 314, 326
 nazwa, 227
 NIO, 300
 Node, 477, 484
 NonIntResultException, 270

klasa

Number, 351
 Object, 225, 329, 394
 metoda, 225
 odmowa dostępu, 269
 opakowująca, 300, 302
 OutputStream, 276, 278, 280, 288
 OutputStreamWriter, 298
 Panel, 432
 Parent, 477, 480, 484
 pochodna, 193, 195, 206
 konstruktor, 198, 199
 metoda, 213
 wyjątek, 259
 PrintStream, 280, 281
 PrintWriter, 297, 298
 Priority, 322
 RandomAccessFile, 291, 292
 Reader, 276, 293, 294, 299
 Region, 477, 484
 RuntimeException, 266
 Scanner, 306
 Scene, 476, 477, 479
 składowa, 98, 178
 chroniona, 230, 232
 kontrola dostępu, 159, 160, 196, 198, 230
 prywatna, 159, 160, 161, 163, 196, 198, 230
 publiczna, 159, 160, 163, 230
 static, 389
 statyczna, 356, 357, 358
 sparametryzowana, 368, 369, 376, 381, 391, 492
 Stage, 476, 479
 String, 141, 144
 StringBuffer, 144
 StringBuilder, 144
 SumArray, 326
 Sync, 326
 System, 235, 277
 TextField, 495
 Thread, 310, 311, 318
 Throwable, 254, 259, 262, 263, 270, 391
 konstruktor, 269
 Transform, 500
 wejścia, 275
 wewnętrzna, 184
 anonimowa, 470, 471
 Writer, 276, 293, 298
 wyjścia, 275
 zagnieżdżona, 184, 186
 static, 186
 w bloku, 184, 185

koder, 300
 kolejka, 134, 241
 cykliczna, 134, 241, 242, 243
 implementacja, 241
 zwykła, 134
 kolekcja, 138
 komentarz, 551, 555
 kompilator, 43, 100, 358
 komponent, 447
 komunikat, 427
 konsoli, 358
 w oknie statusu, 430
 zachęty, 266
 konsola, 277, 293, 445
 wyjście, 298
 znakowa, 67, 275
 konstruktor, 113, 115, 343, 442
 klasy, *Patrz:* klasa konstruktor
 parametr, 114
 przeciążanie, 173, 174, 442
 referencja, 414, 416
 sparametryzowany, 378
 wywoływanie, 208
 kontener, 447, 448
 hierarchia zawierania, 447
 szczytowy, 448
 panel, *Patrz:* panel
 kontroler, 447
 kontrolka, 481, 483
 CheckBox, 487, 488
 efekty, 498, 501
 ListView, 487, 491, 492, 494
 zaznaczanie kilku elementów, 495
 modyfikowanie wyglądu, 498, 500, 501
 PasswordField, 498
 TextArea, 498
 TextField, 487, 495
 transformacja, 498, 500, 501
 korzeń, 477, 480

L

lambda wyrażeniowa, 400, *Patrz też:* wyrażenie
 lambda
 lista, 483, 491, 494
 JList, 462
 zdarzenie, 463
 liść, *Patrz:* węzeł końcowy
 literał, 49, 51 *Patrz też:* stała
 łańcuchowy, *Patrz:* łańcuch znakowy

Ł

łańcuch

- numeryczny, 299
 - zamiana na postać binarną, 306
- znakowy, 50, 141, 269, 462, 482, 495
 - jednoznakowy, 51
 - łączenie, 143
 - modyfikacja, 144
 - operacje, 142
 - tworzenie, 141
 - wczytywanie, 296

M

- maszyna wirtualna, 44, 178, 442
 - błąd, 254
- menedżer układu, 449, 452
 - BorderLayout, 449, 452, 453, 456
 - BoxLayout, 449
 - FlowLayout, 449, 456
 - GridBagLayout, 449
 - GridLayout, 449
 - SpringLayout, 449
 - wykonania, 328
- metadane, *Patrz:* adnotacja
- metoda, 52, 97, 102, 104, 343
 - abstrakcyjna, 220, 235, 246, 394, 395, 397
 - pojedyncza, 394
 - actionPerformed, 454, 456, 457, 459, 470
 - add, 453, 482
 - addActionListener, 457
 - addAll, 486
 - addKeyListener, 435
 - addListener, 493
 - addMouseMotionListener, 435
 - anonimowa, 394
 - argument, *Patrz:* argument
 - change, 492
 - Character, 405
 - charAt, 144
 - clone, 225
 - close, 281, 282, 284, 286, 288
 - compareTo, 302, 345
 - currentThread, 336
 - destroy, 425, 426, 433, 471
 - domyślna, 247, 249, 250, 395
 - dostępowa, 196, 198
 - doubleValue, 351, 352
 - equals, 143, 225
 - finalize, 117, 225
 - funkcyjna, 396
 - get, 134, 135
 - getAccessibleContext, 433
 - getActionCommand, 470
 - getAppletContext, 433
 - getAppletInfo, 433
 - getAudioClip, 433
 - getChildren, 482
 - getClass, 225, 366
 - getCodeBase, 433
 - getContentPane, 452
 - getDocumentBase, 433
 - getErrorMessage, 167
 - getGraphics, 427
 - getImage, 433
 - getLocale, 433
 - getName, 314, 366
 - getParameter, 434
 - getParameterInfo, 434
 - getPriority, 322
 - getSelectedIndex, 463, 465
 - getSelectedItems, 495
 - getSelectionModel, 492, 493
 - getSuppressed, 288
 - getText, 496
 - getValue, 396
 - handle, 484
 - hashCode, 225
 - hasNext, 306
 - init, 425, 426, 434, 471, 477, 478, 480
 - initCause, 270
 - instancyjna
 - referencja, 411
 - interfejsu, 237
 - stacyczna, 251
 - intValue, 352
 - invokeAndWait, 453
 - invokeLater, 453
 - isActive, 434
 - isAlive, 319
 - isEmpty, 483
 - isIndeterminate, 491
 - isSelected, 490
 - isUpperCas, 405
 - isValidRoot, 434
 - itemStateChanged, 460
 - klasy Object, 225
 - launch, 478, 479
 - main, 52, 98
 - Math.abs, 372
 - Math.pow, 356

metoda

- Math.sqrt, 356
- mouseClicked, 437
- mouseEntered, 437
- mouseExited, 437
- mousePressed, 437
- mouseReleased, 437
- natywna, 442
- next, 307
- notify, 225, 329
- notifyAll, 225, 329
- obj.run, 453
- observableArrayList, 494
- ordinal, 345
- paint, 425, 426, 427
- parametr, 106, 107
- parseDouble, 301
- parseInt, 301
- parsująca, 301, 306
- play, 434
- pow, 357
- print, 280, 297
- printf, 293
- println, 46, 50, 275, 280, 297, 299
- printStackTrace, 263
- prompt, 266
- przeciążanie, 169, 170, 172, 189, 190
- przesłanie, 213, 215, 216, 217
- public, 237
- put, 134
- read, 278, 279, 281, 284, 292, 294, 295
- readLine, 293, 296
- referencja, 409, 411, 414
- rekurencyjna, 177, 178
- remove, 483
- removeActionListener, 457
- removeKeyListener, 435
- repaint, 427
- resize, 434
- resume, 333, 334
- rozszerzeń, 247
- run, 311, 334
- seek, 292
- selectedItemProperty, 493
- setActionCommand, 457, 459
- setAllowIndeterminate, 491
- setCharAt, 144
- setDefaultCloseOperation, 451
- setEffect, 498
- setLayout, 452
- setName, 314
- setOnAction, 484
- setPreferredSize, 465
- setPrefSize, 492
- setPriority, 321
- setPromptText, 496
- setSelectionMode, 463, 495
- setStub, 434
- setText, 486
- setVisible, 452
- show, 481
- showAll, 490
- showStatus, 430, 434
- showType, 366
- sleep, 312, 326
- sparametryzowana, 372, 376, 378
 - referencja, 414
- sqrt, 45, 46, 357
- start, 311, 425, 426, 434, 471, 477, 478, 480, 481
- static, 179, 180, 181
- statyczna, 251, 358, 395
 - referencja, 409
- stop, 333, 334, 425, 426, 434, 471, 477, 478
- substring, 145
- sumArray, 326, 327
- suspend, 333, 334
- synchronizacja, 324, 326, 327
- System.console, 293
- toLowerCase, 405
- toString, 225, 263, 297
- toUpperCase, 405
- update, 427
- valueChanged, 462, 463, 465
- valueOf, 342, 343
- values, 342, 343
- valuesOf, 342
- void, 103, 104, 105
- wait, 225, 329
- write, 280, 284, 292
- writeExternal, 554
- writeObject, 554
- wybór dynamiczny, 215
- zwracająca tablicę, 131
- zwracanie obiektów, 167
- metodagetCause, 270
- model, 447
- model-delegat, 447
- model-widok-kontroler, 447
- modyfikator
 - abstract, 220
 - dostępu, 160, 230
 - private, 160, 161, 163, 196, 230
 - protected, 160, 230, 232, 234
 - public, 160, 230

monitor, 324
 operacja niedozwolona, 269
 mysz, 437, 438

N

namiastka, 434
 node, *Patrz:* węzeł

O

obiekt
 AssertionError, 441
 AudioClip, 433, 434
 inicjalizacja, 113
 nasłuchujący, 433, 435, 436, 456, 470, 486, 493
 interfejs, 436
 przekazywanie do metody, 164, 165
 referencja, 101, 119, 209, 210
 pusta, 269
 System.err, 277
 System.in, 277, 278, 279, 293
 System.out, 277, 280, 293, 297, 298
 tworzenie, 99, 101
 zwracany przez metodę, *Patrz:* metoda
 zwracanie obiektów
 obsługa wyjątków, *Patrz:* wyjątek
 obszar roboczy, 476
 opakowywanie, 352
 automatyczne, 351, 352, 353, 354, 356, 367
 operator, 54
 !, 57
 !=, 56
 &, 57, 147
 &&, 57, 58
 *, 54
 ., 99
 /, 54
 ?, 155, 156, 407
 ^, 57, 147
 |, 57, 147
 ||, 57, 58
 ~, 147
 +, 54
 ++, 55
 <, 56
 <<, 151
 <=, 56
 =, 59
 ==, 56, 143
 >, 56

->, 394
 >=, 56
 >>, 151
 >>>, 151
 arytmetyczny, 54, 55, 63
 bitowy, 54, 147
 postać skrócona, 153
 dekrementacji, 55
 diamentowy, 387
 inkrementacji, 55
 konkatencji, 143
 lambda, 394
 logiczny, 54, 56, 57, 63
 warunkowy, 58, 59
 new, 101, 116, 174, 262
 porównania, 56, 143
 priorytet, 63
 przesunięcia, 151
 przypisania, 54, 59, 60
 skrótowy, 59
 złożony, 59
 relacyjny, 54, 56
 ternarny, 155

P

pakiet, 227, 252
 AWT, 422, 425, 432, 447, 449
 domyślny, 228
 hierarchia, 228
 importowanie, 234
 java.awt, *Patrz:* pakiet AWT
 java.awt.event, 435, 436, 456
 java.io, 286
 java.lang, 235, 310, 351, 356, 359, 377
 wyjątek, 268
 java.lang.annotation, 359
 java.nio, 300
 java.util, 306
 java.util.concurrent, 328
 java.util.function, 393, 417
 java.util.stream, 418
 javafx.application, 476, 479
 javafx.beans.value, 492
 javafx.collections, 482, 492
 javafx.event, 484
 javafx.scene, 476, 479
 javafx.scene.control, 484, 487
 javafx.scene.efect, 498
 javafx.scene.layout, 476, 477, 479
 javafx.scene.transform, 500

pakiet

javafx.stage, 476, 479
javafxapplication, 477
javax.swing, 448, 451, 463
javax.swing.event, 462
JDK, 423

kontrola dostępu, 230
lokalizacja, 228
nazwa, 227, 228
prawa dostępu, 228
wyszukiwanie, 228

pamięci odzyskiwanie, 116, 117, 267

panel

szklany, 448, 449
warstw, 448, 449
zawartości, 448, 449

parent node, *Patrz:* węzeł rodzica

pasek przewijania, 494

pętla

bez ciała, 79
do-while, 67, 82, 83, 84, 91
for, 67, 76, 77, 78, 79, 82, 84
 rozszerzona, 80, 137, 138, 139, 140, 141
for-each, 137, 138
nieskończona, 79
przerywanie, 86
while, 67, 81, 84, 91
zagnieżdżanie, 94
zmienna sterująca, 80

plik

.class, 100, 228
dostęp, 291
odczyt, 281
porównywanie, 290, 465
wskaźnik zawartości, 291
zamknięcie, 281, 282, 284, 314
 automatyczne, 285
zapis, 281
zwalnianie, 267, 281, 284

pole tekstowe, 495

komunikat, 496
wielkość, 496

pole wyboru, 487, 488, 490

polimorfizm, 169, 172, 215, 216

potok, 418

primary stage, *Patrz:* obszar roboczy główny

proces, 309

programowanie

równoległe, 328
wielowątkowe, *Patrz:* wielowątkowość

przesłanie, 120, 203, 220, 223

przeźreń nazw, 227, 358

globalna, 358

przycisk, 454, 456, 483, 485

R

rekurencja, 177

root node, *Patrz:* korzeń

rzutowanie, 61, 359, 407

niedozwolone, 269

SSAM, *Patrz:* typ SAM

scena, 476

graf, 477, 481

scene, *Patrz:* scenascene graph, *Patrz:* graf sceny

selektora, 300

serializacja, 552, 554

słowo kluczowe, 102, *Patrz też:* instrukcja

assert, 440, 441

catch, 254, 259, 262, 263, 267, 268

class, 98

enum, 340, 342

extends, 193, 237, 246

final, 223, 224, 225

finally, 254, 265, *Patrz też:* blok finally

implements, 237

import, 356

instanceof, 440, 441

interface, 235

native, 440, 442

protected, 117, 234

static, 178, 356

strictfp, 440, 441

super, 199, 203, 208

synchronized, 324, 326, 327

this, 119, 442

throw, 254

throws, 254, 263, 266, 408

transient, 440

try, 254

void, 102

volatile, 440

słowo kluczowe extends, 370

słuchacz, *Patrz:* obiekt nasłuchujący

sortowanie, 126

pęcherzykowe, 126, 128

Quicksort, 128, 182

stage, *Patrz:* obszar roboczy

stała, 49, 246, *Patrz też*: literal
 wyliczeniowa, 340, 341, 343
 znakowa, 49
 stos, 134, 178
 przepelnienie, 178
 strumień, 276, 418
 bajtowy, 276, 277, 278, 281
 tworzenie, 281
 błędów standardowy, 277
 InputStream, 289
 wejścia standardowy, 277, 279, 293
 wyjścia standardowy, 277, 293, 297, 298
 znakowy, 276, 278, 293, 297, 298
 obiekt, 434
 Swing, 445, 446, 447, 465, 471, 475
 komponent, 446, 448, 450
 metal, 446
 Nimbus, 446
 sygnatura, 173, 235
 synchronizator, 328
 szkielec Fork/Join, 328

T

tablica, 80, 123, 390, 407, 409, 416
 deklaracja, 131, 269
 dwuwymiarowa, 128
 deklaracja, 128
 składowa length, 133
 jednowymiarowa, 123, 124
 deklaracja, 124
 łańcuchów znakowych, 144
 nieregularna, 129, 130
 przekroczenie zakresu, 269
 referencja, 131
 składowa length, 132, 133
 sortowanie, 126
 wielowymiarowa, 128, 139
 deklaracja, 130
 inicjalizacja, 130
 pamięć, 129
 target type, *Patrz*: typ docelowy
 terminal node, *Patrz*: węzeł końcowy
 typ, 43
 bazowy, 124
 boolean, 44, 47
 Boolean, 300, 351
 byte, 44, 45, 49, 147
 Byte, 300, 351
 całkowity, 44, 45
 klasa opakowująca, 300
 char, 44, 45, 46, 49, 147
 Character, 300, 351
 Comparable, 377
 docelowy, 394
 double, 44, 45
 Double, 300, 351
 FileInputStream, 287
 float, 44, 45, 116
 Float, 300, 351
 InputStream, 277
 int, 44, 45, 49, 116, 147
 Integer, 300, 351
 interfejs, 239
 kontrola zgodności, 43, 51, 209
 konwersja
 automatyczna, 60, 170, 171
 rzutowanie, *Patrz*: rzutowanie
 w instrukcji przypisania, 60
 logiczny, 47
 long, 44, 45, 147
 Long, 300, 351
 modyfikator, 440
 MyThread, 331
 nieobiektowy, 44
 numeryczny, 351
 obiektowy, 44
 Object, 297
 ObservableList, 492
 ogólny, 225
 ograniczanie, 370, 371, 380
 opakowujący, 351
 podstawowy, *Patrz*: typ prosty
 PrintStream, 277
 prosty, 44, 49, 116, 165, 351
 klasa opakowująca, 300, 302
 konwersja na obiekt, 354
 prymitywny, *Patrz*: typ prosty
 referencyjny, 165, 166, 367
 SAM, 394
 short, 44, 45, 49, 147
 Short, 300, 351
 sparametryzowany, 363, 364, 367, 384, 387, 388, 391
 ograniczenia, 387, 388, 389, 390
 String, 141
 surowy, 384, 386
 Thread, 313
 wnioskowanie, 387
 wyliczeniowy, 269, 339, 340, 342, 343, 345
 ograniczenia, 345
 zasada promocji, 64, 65
 zgodność, 60, 61, 64
 zmiennoprzecinkowy, 45

U

u, 50

Unicode, 46, 47, 276, 295, 298

W

wartość

porządkowa, 345

ujemna, 152

wątek, 269, 309, 310

aplikacji, 481

fałszywe przebudzenie, 329

główny, 310, 336, 481

gotowy do wykonywania, 310

komunikacja, 328

kończenie działania, 333, 334

priorytet, 321, 323

przełączanie kontekstu, 336

pula, 328

rozdziału zdarzeń, 453, 471

synchronizacja, 310, 324, 333

tworzenie, 310, 311, 315, 317, 318

uruchomieniowy, *Patrz:* wątek główny

wstrzymywanie, 333, 334

wykonywany, 310

wznawianie, 333, 334

zablokowany, 310

zakończony, 310, 319

zawieszony, 310, 329, 333

węzeł, 477

dodawanie, 482

gałęzi, 477

końcowy, 477

potomny, 482

rodzica, 477, 486

transformacja, 500, 501

usuwanie, 483

wygląd, 498

widok, 447

wielowątkowość, 309, 310, 314, 336

zakleszczenie, 333

wielozadaniowość, 309

implementacja, 323

z wywłaszczeniem, 323

współbieżność API, 328

wyjątek, 116, 253, 254, 271, 553

ArithmeticException, 258, 262, 267, 268, 269

ArrayIndexOutOfBoundsException, 126, 255,
257, 258, 261, 267, 268, 269

ArrayStoreException, 269

AssertionError, 441

ClassCastException, 269

ClassNotFoundException, 269

CloneNotSupportedException, 269

EnumConstantNotPresentException, 269

FileNotFoundException, 281, 284, 299

generowanie, 262

IllegalAccessException, 269

IllegalArgumentException, 269

IllegalMonitorStateException, 269

IllegalStateException, 269

IllegalThreadStateException, 269

IndexOutOfBoundsException, 269

InstantiationException, 269

InterruptedException, 269

IOException, 266, 268, 281, 284, 288, 289, 293, 408

jako przyczyna wyjątku, 269

klauzula catch, *Patrz:* słowo kluczowe catch

lista, 267

łańcuch, 269

NegativeArraySizeException, 269

nieprzechwycony, 257

niesprawdzany, 268, 269

NonIntResultException, 270

NoSuchFieldException, 269

NoSuchMethodException, 269

NullPointerException, 269

NumberFormatException, 269

obsługa domyślna, 257

przechwytywanie, 254

ReflectiveOperationException, 269

RuntimeException, 268

SecurityException, 269

sprawdzany, 268, 269

StringIndexOutOfBoundsException, 269

TypeNotPresentException, 269

UnsupportedOperationException, 269

wbudowany w Javę, 268

wyrażenie lambda, *Patrz:* wyrażenie lambda
wyjątek

wyliczenie, 340, 344, 345

wymazywanie, 366, 388

wypakowywanie, 351, 352, 353, 354, 356

wyrażenie, 64

lambda, 17, 393, 394, 396, 397, 407, 418, 470

blokowe, 400, 401

ciało, 394

ciało wyrażeniowe, 400

jako argument, 403, 405

parametr, 396, 400, 401, 409

przechwytywanie zmiennych, 407

referencja konstruktora, 414, 416

referencja metody, 409, 411, 414

wyjątek, 408

Z

- zasada promocji typów, 64, 65
- zawieranie, 447
- zbiór znaków, *Patrz:* znak zbiór
- zdarzenie, 421, 424, 433, 435, 436, 460, 483, 485
 - ActionEvent, 484, 486, 496
 - ListSelectionEvent, 462
 - model delegacji, 433, 435, 436
 - myszy, *Patrz:* mysz
 - źródło, 435
- zmienna
 - CLASSPATH, 229
 - deklarowanie, 51, 53
 - final, 342, 345
 - inicjalizacja, 51, 53
 - interfejsu, 245
 - iteracyjna, 137
 - lokalna, 52
 - finalizowanie, 407
 - przechwytywanie, 407
 - referencyjna, 209, 210, 215
 - static, 181
 - sterująca pętlą, *Patrz:* pętla zmienna sterująca
 - zasięg, 52, 53
- znacznik
 - @author, 552
 - @code, 552
 - @deprecated, 552
 - @docRoot, 552, 553
 - @exception, 552, 553
 - @inheritDoc, 552, 553
 - @link, 552, 553
 - @linkplain, 552, 553
 - @literal, 552, 553
 - @param, 552, 553
 - @return, 552, 553
 - @see, 552, 554, 555
 - @serial, 552, 554
 - @serialData, 552, 554
 - @serialField, 552, 554
 - @since, 552, 554
 - @throws, 552, 554
 - @value, 552, 554
 - @version, 552, 555
 - javadoc, 551, 552
- znak, 46
 - \, 50
 - !, 57
 - !=", 56
 - ", 50
 - ", 50
 - &, 57, 147
 - &&, 58
 - *, 54
 - */, 551
 - /, 54
 - /*, 551
 - /**, 551, 555
 - //, 551
 - ?, 155, 156, 407
 - @, 551, 555
 - \\, 50
 - ^, 57, 147
 - |, 57, 147
 - ||, 58
 - ~, 147
 - +, 54
 - ++, 55
 - <, 56
 - <<, 151
 - <=, 56
 - =, 59
 - ==, 56
 - >, 56
 - >, 394
 - >=, 56
 - >>, 151
 - >>>, 151
 - ASCII, *Patrz:* ASCII
 - \b, 50
 - \f, 50
 - kropki, 99
 - lewego ukośnika, 50
 - łańcuch, *Patrz:* łańcuch znakowy
 - \n, 50
 - podkreślenia, 49
 - \r, 50
 - sekwencja specjalna, *Patrz:* sekwencja specjalna
 - \t, 50
 - Unicode, *Patrz:* Unicode
 - wprowadzanie z klawiatury, 67
 - zbiór, 300

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

Poznaj i wykorzystaj możliwości języka Java!

Java to najczęściej wybierany język programowania do zadań specjalnych. Jeżeli Twój projekt wymaga najwyższego poziomu bezpieczeństwa, ogromnej wydajności oraz sprawdzonych rozwiązań, to wybór może być tylko jeden! Właśnie Java jest najczęściej używana do tworzenia zaawansowanych systemów bankowych oraz aplikacji do zarządzania przedsiębiorstwami i finansami. Ale czy sprawdzi się w typowym projekcie? Oczywiście!

Rozpocznij własną przygodę z tym popularnym językiem programowania korzystając z tej książki. Kolejne wydanie zostało poprawione i zaktualizowane o nowości ze świata Javy. Każda strona zawiera bezcenną wiedzę na temat składni języka, stosowanych w nim konstrukcji, programowania obiektowego i nie tylko. Sprawdzisz tu, jak obsługiwać wyjątki, korzystać ze strumieni oraz z wątków. Jeżeli masz ambicję stworzyć atrakcyjny interfejs użytkownika z użyciem JavaFX, również będziesz usatysfakcjonowany znalezionymi w tej książce informacjami. Jest to doskonała lektura dla wszystkich osób chcących poznać Javę — jeden z najbardziej cenionych języków programowania.

Dzięki tej książce:

- poznasz składnię oraz typowe konstrukcje języka Java
- zdobędziesz wiedzę na temat programowania obiektowego
- obsłużysz sytuacje wyjątkowe w Twojej aplikacji
- wykorzystasz możliwości JavaFX
- opanujesz cenną wiedzę o języku programowania

Herbert Schildt — autorytet w świecie programistów Javy, C, C++ oraz C#. Jest autorem bestsellerów poświęconych programowaniu w tych językach. Swoją uwagę koncentruje na językach programowania, interpreterach, kompilatorach oraz sterowaniu robotami. Bierze aktywny udział w procesie standaryzacji języków programowania.

Helion	
34430 numer katalogowy	Sprawdź najnowsze promocje: • http://helion.pl/promocje Książki najchętniej czytane: • http://helion.pl/bestsellery Zamów informacje o nowościach: • http://helion.pl/novosci
księgarnia internetowa	
http://helion.pl	
zamówienia telefoniczne	
0 801 339900	Helion SA ul. Kościuszki 1c, 44-100 Gliwice tel.: 32 230 98 63 e-mail: helion@helion.pl http://helion.pl
0 601 339900	
Informatyka w najlepszym wydaniu	



ISBN 978-83-283-0808-4

9 788328 308084

cena: 89,00 zł

