

Zdobądź wiedzę i umiejętności, które pomogą Ci stworzyć efektywne aplikacje dla systemu Android!



# Java

## Przygotowanie do programowania na platformę Android

**Jeff Friesen**

Odkryj Javę — od podstaw po zaawansowane mechanizmy tego języka

Dowiedz się, jak wykorzystać potencjał API platformy

Poznaj wszystkie aspekty programowania niezbędne do tworzenia aplikacji na urządzenia przenośne

Apress®



Tytuł oryginału: Learn Java for Android Development

Tłumaczenie: Daniel Kaczmarek (wstęp, rozdz. 1 – 7, dod. A)  
Aleksander Lamża (rozdz. 8 – 10, dod. A)

ISBN: 978-83-246-3372-2

Original edition copyright © 2010 by Jeff “JavaJeff” Friesen.  
All rights reserved.

Polish edition copyright © 2012 by Helion S.A.  
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.

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

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/jappan>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

# Spis treści

<b>O autorze</b> .....	<b>11</b>
<b>O recenzencie technicznym</b> .....	<b>12</b>
<b>Wprowadzenie</b> .....	<b>13</b>
<b>Rozdział 1. Pierwsze kroki w języku Java</b> .....	<b>17</b>
Czym jest Java? .....	17
Java jest językiem programowania .....	18
Java jest platformą .....	19
Java SE, Java EE, Java ME i Android .....	21
Instalacja i poznawanie możliwości JDK .....	22
Instalacja i poznawanie możliwości dwóch najpopularniejszych środowisk IDE .....	27
Zintegrowane środowisko programistyczne NetBeans .....	28
Zintegrowane środowisko programistyczne Eclipse .....	32
Gra karciana Karetą .....	35
Reguły gry w Karetę .....	36
Model gry Karetą w pseudokodzie .....	36
Przekształcenie pseudokodu na kod języka Java .....	38
Kompilowanie, uruchamianie i udostępnianie aplikacji FourOfAKind .....	51
Podsumowanie .....	55
<b>Rozdział 2. Podstawy języka Java</b> .....	<b>57</b>
Klasy .....	57
Deklarowanie klas .....	58
Pola .....	59
Metody .....	73
Konstruktory .....	91
Inne konstrukcje inicjalizujące .....	93
Interfejs a implementacja .....	98
Obiekty .....	102
Tworzenie obiektów i tablic .....	102
Uzyskiwanie dostępu do pól .....	104
Wywoływanie metod .....	106
Odśmianie .....	109
Podsumowanie .....	111

<b>Rozdział 3.</b>	<b>Mechanizmy języka zorientowane obiektowo .....</b>	<b>115</b>
	Dziedziczenie .....	115
	Rozszerzanie klas .....	116
	Najwyższa klasa przodka .....	121
	Kompozycja .....	130
	Problemy z dziedziczeniem implementacji .....	130
	Wielopostaciowość .....	134
	Rzutowanie w górę i późne wiązanie .....	135
	Klasy i metody abstrakcyjne .....	138
	Rzutowanie w dół i identyfikacja typów w fazie wykonania .....	140
	Kowariantne typy zwracanych wartości .....	142
	Interfejsy .....	144
	Deklarowanie interfejsów .....	144
	Implementowanie interfejsów .....	145
	Rozszerzanie interfejsów .....	149
	Po co używać interfejsów? .....	150
	Podsumowanie .....	156
<b>Rozdział 4.</b>	<b>Zaawansowane mechanizmy języka — część I .....</b>	<b>157</b>
	Typy zagnieźdzone .....	157
	Styczne klasy składowe .....	157
	Niestyczne klasy składowe .....	160
	Klasy anonimowe .....	164
	Klasy lokalne .....	166
	Interfejsy wewnątrz klas .....	168
	Pakiety .....	169
	Czym są pakiety? .....	169
	Instrukcja pakietu .....	171
	Instrukcja importu .....	171
	Wyszukiwanie pakietów i typów .....	172
	Korzystanie z pakietów .....	174
	Pakiety i pliki JAR .....	178
	Importy statyczne .....	178
	Wyjątki .....	180
	Czym są wyjątki? .....	181
	Reprezentowanie wyjątków w kodzie źródłowym .....	181
	Rzucanie wyjątków .....	185
	Obsługa wyjątków .....	188
	Wykonywanie czynności sprzątających .....	192
	Podsumowanie .....	198
<b>Rozdział 5.</b>	<b>Zaawansowane mechanizmy języka — część II .....</b>	<b>199</b>
	Asercje .....	199
	Deklarowanie asercji .....	200
	Korzystanie z asercji .....	201
	Unikanie korzystania z asercji .....	207
	Włączanie i wyłączanie asercji .....	207

Adnotacje .....	208
Działanie adnotacji .....	209
Deklarowanie typów adnotacji i wstawianie adnotacji do kodu źródłowego .....	212
Przetwarzanie adnotacji .....	216
Mechanizmy ogólne .....	218
Kolekcje i potrzeba bezpieczeństwa typologicznego .....	219
Typy ogólne .....	221
Metody ogólne .....	232
Typy wyliczeniowe .....	233
Problem z tradycyjnymi typami wyliczeniowymi .....	234
Enum — alternatywa dla tradycyjnego typu wyliczeniowego .....	235
Klasa Enum .....	240
Podsumowanie .....	244
<b>Rozdział 6. Podstawowe interfejsy API — część I .....</b>	<b>247</b>
Interfejsy API do wykonywania obliczeń matematycznych .....	247
Klasy Math i StrictMath .....	247
Klasa BigDecimal .....	254
Klasa BigInteger .....	259
Informacje na temat pakietów .....	263
Podstawowe klasy opakowujące .....	267
Klasa Boolean .....	268
Klasa Character .....	270
Klasy Float i Double .....	271
Klasy Integer, Long, Short i Byte .....	275
Klasa Number .....	277
API References .....	277
Podstawowe pojęcia .....	277
Klasy Reference i ReferenceQueue .....	279
Klasa SoftReference .....	280
Klasa WeakReference .....	283
Klasa PhantomReference .....	284
Podsumowanie .....	288
<b>Rozdział 7. Podstawowe interfejsy API — część II .....</b>	<b>289</b>
API Reflection .....	289
Zarządzanie ciągami znaków .....	297
Klasa String .....	298
Klasa StringBuffer .....	301
Klasa System .....	304
API Threading .....	307
Interfejs Runnable i klasa Thread .....	307
Synchronizacja wątków .....	317
Podsumowanie .....	333
<b>Rozdział 8. Biblioteka kolekcji .....</b>	<b>335</b>
Przegląd biblioteki .....	335
Interfejs Comparable kontra Comparator .....	336

Interfejsy Iterable i Collection .....	338
Iterator i nowa pętla for .....	341
Automatyczne pakowanie i rozpakowywanie .....	342
Interfejs List .....	344
Klasa ArrayList .....	348
Klasa LinkedList .....	349
Interfejs Set .....	351
Klasa TreeSet .....	351
Klasa HashSet .....	353
Klasa EnumSet .....	356
Interfejs SortedSet .....	358
Interfejs Queue .....	365
Klasa PriorityQueue .....	366
Interfejs Map .....	369
Klasa TreeMap .....	373
HashMap .....	374
Klasa IdentityHashMap .....	380
Klasa WeakHashMap .....	382
Klasa EnumMap .....	383
Interfejs SortedMap .....	384
Narzędzia .....	387
Klasyczne klasy kolekcji .....	389
Podsumowanie .....	396
<b>Rozdział 9. Dodatkowe biblioteki klas narzędziowych .....</b>	<b>397</b>
Narzędzia wspomagające współbieżność .....	397
Wykonawcy .....	397
Synchronizatory .....	406
Współbieżne kolekcje .....	408
Blokady .....	410
Zmienne atomowe .....	413
Internacjonalizacja .....	414
Lokalizatory .....	414
Paczki zasobów .....	416
Iteratory operujące na tekście .....	425
Porównywanie tekstów — klasa Collator .....	429
Daty, strefy czasowe i kalendarze .....	430
Formatery .....	436
Biblioteka klas preferencji .....	443
Generowanie liczb pseudolosowych .....	446
Wyrażenia regularne .....	449
Podsumowanie .....	460
<b>Rozdział 10. Operacje wejścia-wyjścia .....</b>	<b>463</b>
Klasa File .....	463
Klasa RandomAccessFile .....	474
Strumienie .....	485
Przegląd klas strumieni .....	485
Klasy OutputStream i InputStream .....	487

Klasy ByteArrayOutputStream i ByteArrayInputStream .....	489
Klasy FileOutputStream i FileInputStream .....	491
Klasy PipedOutputStream i PipedInputStream .....	494
Klasy FilterOutputStream i FilterInputStream .....	497
Klasy BufferedOutputStream i BufferedInputStream .....	504
Klasy DataOutputStream i DataInputStream .....	505
Serializacja i deserializacja obiektów .....	508
Klasa PrintStream .....	519
Klasy Writer i Reader .....	523
Przegląd klas Writer i Reader .....	524
Klasy bazowe Writer i Reader .....	524
Klasy OutputStreamWriter i InputStreamReader .....	525
Klasy FileWriter i FileReader .....	529
Podsumowanie .....	540
Na tym nie koniec .....	541
<b>Dodatek A</b>	
<b>Odpowiedzi do ćwiczeń .....</b>	<b>543</b>
Rozdział 1. Pierwsze kroki w języku Java .....	543
Rozdział 2. Podstawy języka Java .....	548
Rozdział 3. Mechanizmy języka zorientowane obiektowo .....	551
Rozdział 4. Zaawansowane mechanizmy języka — część I .....	558
Rozdział 5. Zaawansowane mechanizmy języka — część II .....	564
Rozdział 6. Podstawowe interfejsy API — część I .....	569
Rozdział 7. Podstawowe interfejsy API — część II .....	572
Rozdział 8. Biblioteka kolekcji .....	578
Rozdział 9. Dodatkowe biblioteki klas narzędziowych .....	583
Rozdział 10. Operacje wejścia-wyjścia .....	589
<b>Skorowidz .....</b>	<b>601</b>





## ROZDZIAŁ 6

# Podstawowe interfejsy API — część I

Poważni programiści aplikacji dla systemu Android muszą gruntownie znać najważniejsze interfejsy API języka Java. Z kilkoma API mieliśmy już do czynienia, dość wspomnieć klasy `Object` i `String` oraz hierarchię klas `Throwable`. W tym rozdziale przedstawimy kolejne podstawowe interfejsy API przeznaczone do wykonywania obliczeń matematycznych, operowania na pakietach i typach podstawowych, a także mechanizm odśmiecania.

---

■ **Uwaga** • W rozdziale 6. zostaną opisane podstawowe klasy i interfejsy API zlokalizowane w pakietach `java.lang`, `java.lang.ref` i `java.math`.

---

## Interfejsy API do wykonywania obliczeń matematycznych

W rozdziale 2. zaprezentowano operatory `+`, `-`, `*`, `/` i `%` języka Java przeznaczone do wykonywania najważniejszych operacji matematycznych na wartościach typów podstawowych. Java udostępnia także klasy przeznaczone do wykonywania operacji trygonometrycznych i innych zaawansowanych działań, precyzyjnej prezentacji wartości pieniężnych oraz obsługi bardzo długich liczb całkowitych wykorzystywanych do szyfrowania algorytmem RSA ([http://pl.wikipedia.org/wiki/RSA\\_\(kryptografia\)](http://pl.wikipedia.org/wiki/RSA_(kryptografia))) i w innych kontekstach.

### Klasy `Math` i `StrictMath`

Klasa `java.lang.Math` deklaruje stałe typu `double` o nazwach `E` i `PI`, które reprezentują odpowiednio podstawę logarytmu naturalnego (2,71828...) oraz stosunek obwodu okręgu do jego średnicy (3,141519...). Stała `E` jest inicjalizowana wartością 2,718281828459045, natomiast stała `PI` ma wartość 3,141592653589793. W klasie `Math` zadeklarowane są także wybrane metody klasy, przeznaczone do wykonywania różnego rodzaju działań matematycznych. W tabeli 6.1 opisano większość dostępnych metod.

**Tabela 6.1.** Metody klasy *Math*

Metoda	Opis
<code>double abs(double d)</code>	Zwraca wartość bezwzględną liczby <i>d</i> . Istnieją cztery przypadki szczególne: $\text{abs}(-0.0) = +0.0$ , $\text{abs}(+\text{nieskończoność}) = +\text{nieskończoność}$ , $\text{abs}(-\text{nieskończoność}) = +\text{nieskończoność}$ oraz $\text{abs}(\text{NaN}) = \text{NaN}$ .
<code>float abs(float f)</code>	Zwraca wartość bezwzględną liczby <i>f</i> . Istnieją cztery przypadki szczególne: $\text{abs}(-0.0) = +0.0$ , $\text{abs}(+\text{nieskończoność}) = +\text{nieskończoność}$ , $\text{abs}(-\text{nieskończoność}) = +\text{nieskończoność}$ oraz $\text{abs}(\text{NaN}) = \text{NaN}$ .
<code>int abs(int i)</code>	Zwraca wartość bezwzględną liczby <i>i</i> . Istnieje jeden przypadek szczególny: wartością bezwzględną <code>Integer.MIN_VALUE</code> jest <code>Integer.MIN_VALUE</code> .
<code>long abs(long l)</code>	Zwraca wartość bezwzględną liczby <i>l</i> . Istnieje jeden przypadek szczególny: wartością bezwzględną <code>Long.MIN_VALUE</code> jest <code>Long.MIN_VALUE</code> .
<code>double acos(double d)</code>	Zwraca arcus cosinus kąta <i>d</i> z przedziału od 0 do $\pi$ . Istnieją trzy przypadki szczególne: $\text{acos}(\text{wartość} > 1) = \text{NaN}$ , $\text{acos}(\text{wartość} < -1) = \text{NaN}$ oraz $\text{acos}(\text{NaN}) = \text{NaN}$ .
<code>double asin(double d)</code>	Zwraca arcus sinus kąta <i>d</i> z przedziału od $-\pi/2$ do $\pi/2$ . Istnieją trzy przypadki szczególne: $\text{asin}(\text{wartość} > 1) = \text{NaN}$ , $\text{asin}(\text{wartość} < -1) = \text{NaN}$ oraz $\text{asin}(\text{NaN}) = \text{NaN}$ .
<code>double atan(double d)</code>	Zwraca arcus tangens kąta <i>d</i> z przedziału $-\pi/2$ do $\pi/2$ . Istnieje pięć przypadków szczególnych: $\text{atan}(+0.0) = +0.0$ , $\text{atan}(-0.0) = -0.0$ , $\text{atan}(+\text{nieskończoność}) = +\pi/2$ , $\text{atan}(-\text{nieskończoność}) = -\pi/2$ oraz $\text{atan}(\text{NaN}) = \text{NaN}$ .
<code>double ceil(double d)</code>	Zwraca najmniejszą wartość (najbliższą minus nieskończoności), która nie jest mniejsza od <i>d</i> i jest liczbą całkowitą. Istnieje sześć przypadków szczególnych: $\text{ceil}(+0.0) = +0.0$ , $\text{ceil}(-0.0) = -0.0$ , $\text{ceil}(\text{wartość} > -1.0 \text{ oraz } < 0.0) = -0.0$ , $\text{ceil}(+\text{nieskończoność}) = +\text{nieskończoność}$ , $\text{ceil}(-\text{nieskończoność}) = -\text{nieskończoność}$ oraz $\text{ceil}(\text{NaN}) = \text{NaN}$ .
<code>double cos(double d)</code>	Zwraca cosinus kąta <i>d</i> (wyrażonego w radianach). Istnieją trzy przypadki szczególne: $\text{cos}(+\text{nieskończoność}) = \text{NaN}$ , $\text{cos}(-\text{nieskończoność}) = \text{NaN}$ oraz $\text{cos}(\text{NaN}) = \text{NaN}$ .
<code>double exp(double d)</code>	Zwraca liczbę Eulera <i>e</i> podniesioną do potęgi <i>d</i> . Istnieją trzy przypadki szczególne: $\text{exp}(+\text{nieskończoność}) = +\text{nieskończoność}$ , $\text{exp}(-\text{nieskończoność}) = +0.0$ oraz $\text{exp}(\text{NaN}) = \text{NaN}$ .
<code>double floor(double d)</code>	Zwraca największą wartość (najbliższą plus nieskończoności), która nie jest większa od <i>d</i> i jest liczbą całkowitą. Istnieje pięć przypadków szczególnych: $\text{floor}(+0.0) = +0.0$ , $\text{floor}(-0.0) = -0.0$ , $\text{floor}(+\text{nieskończoność}) = +\text{nieskończoność}$ , $\text{floor}(-\text{nieskończoność}) = -\text{nieskończoność}$ oraz $\text{floor}(\text{NaN}) = \text{NaN}$ .
<code>double log(double d)</code>	Zwraca logarytm naturalny (przy podstawie <i>e</i> ) z liczby <i>d</i> . Istnieje sześć przypadków szczególnych: $\text{log}(+0.0) = -\text{nieskończoność}$ , $\text{log}(-0.0) = -\text{nieskończoność}$ , $\text{log}(\text{wartość} < 0) = \text{NaN}$ , $\text{log}(+\text{nieskończoność}) = +\text{nieskończoność}$ , $\text{log}(-\text{nieskończoność}) = \text{NaN}$ oraz $\text{log}(\text{NaN}) = \text{NaN}$ .

**Tabela 6.1.** Metody klasy *Math* — ciąg dalszy

Metoda	Opis
<code>double log10(double d)</code>	Zwraca logarytm przy podstawie 10 z liczby <i>d</i> . Istnieje sześć przypadków szczególnych: $\log_{10}(+0.0) = -\text{nieskończoność}$ , $\log_{10}(-0.0) = -\text{nieskończoność}$ , $\log_{10}(\text{wartość} < 0) = \text{NaN}$ , $\log_{10}(\text{+nieskończoność}) = +\text{nieskończoność}$ , $\log_{10}(\text{-nieskończoność}) = \text{NaN}$ oraz $\log_{10}(\text{NaN}) = \text{NaN}$ .
<code>double max(double d1, double d2)</code>	Zwraca największą (najbliższą plus nieskończoności) spośród liczb <i>d1</i> i <i>d2</i> . Istnieją cztery przypadki szczególne: $\max(\text{NaN}, \text{wartość}) = \text{NaN}$ , $\max(\text{wartość}, \text{NaN}) = \text{NaN}$ , $\max(+0.0, -0.0) = +0.0$ oraz $\max(-0.0, +0.0) = +0.0$ .
<code>float max(double f1, double f2)</code>	Zwraca największą (najbliższą plus nieskończoności) spośród liczb <i>f1</i> i <i>f2</i> . Istnieją cztery przypadki szczególne: $\max(\text{NaN}, \text{wartość}) = \text{NaN}$ , $\max(\text{wartość}, \text{NaN}) = \text{NaN}$ , $\max(+0.0, -0.0) = +0.0$ oraz $\max(-0.0, +0.0) = +0.0$ .
<code>int max(int i1, int i2)</code>	Zwraca największą (najbliższą plus nieskończoności) spośród liczb <i>i1</i> i <i>i2</i> .
<code>long max(long l1, long l2)</code>	Zwraca największą (najbliższą plus nieskończoności) spośród liczb <i>l1</i> i <i>l2</i> .
<code>double min(double d1, double d2)</code>	Zwraca najmniejszą (najbliższą minus nieskończoności) spośród liczb <i>d1</i> i <i>d2</i> . Istnieją cztery przypadki szczególne: $\min(\text{NaN}, \text{wartość}) = \text{NaN}$ , $\min(\text{wartość}, \text{NaN}) = \text{NaN}$ , $\min(+0.0, -0.0) = -0.0$ oraz $\min(-0.0, +0.0) = -0.0$ .
<code>float min(float f1, float f2)</code>	Zwraca najmniejszą (najbliższą minus nieskończoności) spośród liczb <i>f1</i> i <i>f2</i> . Istnieją cztery przypadki szczególne: $\min(\text{NaN}, \text{wartość}) = \text{NaN}$ , $\min(\text{wartość}, \text{NaN}) = \text{NaN}$ , $\min(+0.0, -0.0) = -0.0$ oraz $\min(-0.0, +0.0) = -0.0$ .
<code>int min(int i1, int i2)</code>	Zwraca najmniejszą (najbliższą minus nieskończoności) spośród liczb <i>i1</i> i <i>i2</i> .
<code>long min(long l1, long l2)</code>	Zwraca najmniejszą (najbliższą minus nieskończoności) spośród liczb <i>l1</i> i <i>l2</i> .
<code>double random()</code>	Zwraca liczbę pseudolosową z przedziału prawostronnie otwartego od 0,0 (włącznie) do 1,0.
<code>long round(double d)</code>	Zwraca wynik zaokrąglenia liczby <i>d</i> do długiej liczby całkowitej. Wynik jest równoważny wynikowi wyrażenia $(\text{long}) \text{Math.floor}(d+0.5)$ . Istnieje siedem przypadków szczególnych: $\text{round}(+0.0) = +0.0$ , $\text{round}(-0.0) = +0.0$ , $\text{round}(\text{wartość} > \text{Long.MAX\_VALUE}) = \text{Long.MAX\_VALUE}$ , $\text{round}(\text{wartość} < \text{Long.MIN\_VALUE}) = \text{Long.MIN\_VALUE}$ , $\text{round}(+\text{nieskończoność}) = \text{Long.MAX\_VALUE}$ , $\text{round}(-\text{nieskończoność}) = \text{Long.MIN\_VALUE}$ oraz $\text{round}(\text{NaN}) = +0.0$ .
<code>int round(float f)</code>	Zwraca wynik zaokrąglenia liczby <i>f</i> do liczby całkowitej. Wynik jest równoważny wynikowi wyrażenia $(\text{long}) \text{Math.floor}(f+0.5)$ . Istnieje siedem przypadków szczególnych: $\text{round}(+0.0) = +0.0$ , $\text{round}(-0.0) = +0.0$ , $\text{round}(\text{wartość} > \text{Integer.MAX\_VALUE}) = \text{Integer.MAX\_VALUE}$ , $\text{round}(\text{wartość} < \text{Integer.MIN\_VALUE}) = \text{Integer.MIN\_VALUE}$ , $\text{round}(+\text{nieskończoność}) = \text{Integer.MAX\_VALUE}$ , $\text{round}(-\text{nieskończoność}) = \text{Integer.MIN\_VALUE}$ oraz $\text{round}(\text{NaN}) = +0.0$ .
<code>double signum(double d)</code>	Zwraca znak liczby <i>d</i> jako liczbę $-1.0$ (jeżeli <i>d</i> jest mniejsze od 0,0), liczbę $0.0$ (jeżeli <i>d</i> jest równe 0,0) lub $1.0$ (jeżeli <i>d</i> jest większe niż 0,0). Istnieje pięć przypadków szczególnych: $\text{signum}(+0.0) = +0.0$ , $\text{signum}(-0.0) = -0.0$ , $\text{signum}(+\text{nieskończoność}) = +1.0$ , $\text{signum}(-\text{nieskończoność}) = -1.0$ oraz $\text{signum}(\text{NaN}) = \text{NaN}$ .

**Tabela 6.1.** Metody klasy *Math* — ciąg dalszy

Metoda	Opis
<code>float signum(float f)</code>	Zwraca znak liczby <i>f</i> jako liczbę $-1,0$ (jeżeli <i>f</i> jest mniejsze od $0,0$ ), liczbę $0,0$ (jeżeli <i>f</i> jest równe $0,0$ ) lub $1,0$ (jeżeli <i>f</i> jest większe niż $0,0$ ). Istnieje pięć przypadków szczególnych: $\text{signum}(+0.0) = +0.0$ , $\text{signum}(-0.0) = -0.0$ , $\text{signum}(+\text{nieskończoność}) = +1.0$ , $\text{signum}(-\text{nieskończoność}) = -1.0$ oraz $\text{signum}(\text{NaN}) = \text{NaN}$ .
<code>double sin(double d)</code>	Zwraca sinus kąta <i>d</i> (wyrażonego w radianach). Istnieje pięć przypadków szczególnych: $\text{sin}(+0.0) = +0.0$ , $\text{sin}(-0.0) = -0.0$ , $\text{sin}(+\text{nieskończoność}) = \text{NaN}$ , $\text{sin}(-\text{nieskończoność}) = \text{NaN}$ oraz $\text{sin}(\text{NaN}) = \text{NaN}$ .
<code>double sqrt(double d)</code>	Zwraca pierwiastek kwadratowy liczby <i>d</i> . Istnieje pięć przypadków szczególnych: $\text{sqrt}(+0.0) = +0.0$ , $\text{sqrt}(-0.0) = -0.0$ , $\text{sqrt}(\text{wartość} < 0) = \text{NaN}$ , $\text{sqrt}(+\text{nieskończoność}) = +\text{nieskończoność}$ oraz $\text{sqrt}(\text{NaN}) = \text{NaN}$ .
<code>double tan(double d)</code>	Zwraca tangens kąta <i>d</i> (wyrażonego w radianach). Istnieje pięć przypadków szczególnych: $\text{tan}(+0.0) = +0.0$ , $\text{tan}(-0.0) = -0.0$ , $\text{tan}(+\text{nieskończoność}) = \text{NaN}$ , $\text{tan}(-\text{nieskończoność}) = \text{NaN}$ oraz $\text{tan}(\text{NaN}) = \text{NaN}$ .
<code>double toDegrees (double angrad)</code>	Przekształca miarę kąta <i>angrad</i> z radianów na stopnie za pomocą wyrażenia $\text{angrad} * 180 / \text{PI}$ . Istnieje pięć przypadków szczególnych: $\text{toDegrees}(+0.0) = +0.0$ , $\text{toDegrees}(-0.0) = -0.0$ , $\text{toDegrees}(+\text{nieskończoność}) = +\text{nieskończoność}$ , $\text{toDegrees}(-\text{nieskończoność}) = -\text{nieskończoność}$ oraz $\text{toDegrees}(\text{NaN}) = \text{NaN}$ .
<code>double toRadians (angdeg)</code>	Przekształca miarę kąta <i>angdeg</i> ze stopni na radiany za pomocą wyrażenia $\text{angdeg} / 180 * \text{PI}$ . Istnieje pięć przypadków szczególnych: $\text{toRadians}(+0.0) = +0.0$ , $\text{toRadians}(-0.0) = -0.0$ , $\text{toRadians}(+\text{nieskończoność}) = +\text{nieskończoność}$ , $\text{toRadians}(-\text{nieskończoność}) = -\text{nieskończoność}$ oraz $\text{toRadians}(\text{NaN}) = \text{NaN}$ .

W tabeli 6.1 przedstawiono obszerny zbiór metod przydatnych do wykonywania działań matematycznych. Na przykład każda metoda `abs` zwraca **wartość bezwzględna** (czyli liczbę bez względu na znak) przekazanego do niej argumentu.

Metody `abs(double)` oraz `abs(float)` przydają się do bezpiecznego porównywania liczb zmiennopozycyjnych o podwójnej precyzji oraz liczb zmiennopozycyjnych. Na przykład wyrażenie `0.3 == 0.1+0.1+0.1` ma wartość `false`, ponieważ liczba `0,1` nie ma dokładnej reprezentacji. Wyrażenia te można jednak ze sobą porównać przy użyciu metody `abs()` i wartości tolerancji, która wskazuje akceptowalny poziom błędu. Na przykład wyrażenie `Math.abs(0.3 - (0.1+0.1+0.1)) < 0.1` będzie już mieć wartość `true`, ponieważ bezwzględna różnica między `0.3` i `0.1+0.1+0.1` jest mniejsza niż wartość tolerancji `0,1`.

We wcześniejszych rozdziałach przedstawiono inne metody klasy *Math*. Na przykład w rozdziale 2. zostały wykorzystane metody `sin()`, `toRadians()`, `cos()`, `round(double)` i `random()` tej klasy.

Jak widać na przykładzie aplikacji *Lotto649* z rozdziału 5., funkcja `random()` (która zwraca liczbę wyglądającą na losową, choć w rzeczywistości jest wyznaczana przez określoną funkcję matematyczną i dlatego jest tak naprawdę **liczbą pseudolosową**) przydaje się do symulacji, gier i w innych zastosowaniach, w których potrzeba losowości. Najpierw jednak

liczbę zwracaną przez `random()`, która należy do przedziału od 0,0 do prawie 1,0, trzeba ją coś przekształcić do wartości bardziej przydatnej, na przykład należącej do przedziału od 0 do 49 albo od -100 do 100. Na listingu 6.1 znajduje się metoda `rnd()`, przydatna do wykonywania tego typu przekształceń.

**Listing 6.1.** Przekształcanie wartości zwracanej przez `random()` do bardziej przydatnej wartości

```
public static int rnd(int limit)
{
    return (int) (Math.random()*limit);
}
```

Metoda `rnd()` przekształca zwracaną przez `random()` liczbę zmiennopozycyjną o podwójnej precyzji z przedziału od 0,0 do 1,0 do liczby całkowitej z przedziału od 0 do `limit - 1`. Na przykład `rnd(50)` zwróci liczbę całkowitą z przedziału od 0 do 49. Z kolei instrukcja `-100+rnd(201)` przekształci przedział od 0,0 do prawie 1,0 do przedziału od -100 do 100 przez dodanie odpowiedniej wartości przesunięcia i wykorzystanie odpowiedniej wartości `limit`.

---

■ **Ostrzeżenie** • Nie należy wykonywać instrukcji `(int) Math.random()*limit`, ponieważ wyrażenie to zawsze będzie mieć wartość 0. W wyrażeniu ułamkowa liczba zmiennopozycyjna o podwójnej precyzji z przedziału od 0,0 do 0,99999... najpierw jest rzutowana na liczbę całkowitą 0 przez ucięcie części ułamkowej, a następnie 0 jest mnożone przez `limit`, co w efekcie daje również 0.

---

W tabeli 6.1 opisano także przypadki szczególne, dotyczące najczęściej wartości +nieskończoność, -nieskończoność, +0.0, -0.0 i NaN (ang. *Not a Number* — wartość, która nie jest liczbą).

Wynikiem obliczeń zmiennopozycyjnych wykonywanych w języku Java mogą być wartości +nieskończoność, -nieskończoność, +0.0, -0.0 i NaN, ponieważ Java w dużej mierze jest zgodna ze standardem IEEE 754 ([http://pl.wikipedia.org/wiki/IEEE\\_754](http://pl.wikipedia.org/wiki/IEEE_754)), który opisuje sposób wykonywania obliczeń zmiennopozycyjnych. Poniżej przedstawiono okoliczności, w których pojawiają się wspomniane wartości specjalne:

- +nieskończoność jest zwracana jako wynik dzielenia liczby dodatniej przez 0,0. Na przykład instrukcja `System.out.println(1.0/0.0)`; zwróci wartość `Infinity`.
- -nieskończoność jest zwracana jako wynik dzielenia liczby ujemnej przez 0,0. Na przykład instrukcja `System.out.println(-1.0/0.0)`; zwróci wartość `-Infinity`.
- NaN jest zwracana jako wynik dzielenia 0,0 przez 0,0, wartość pierwiastka kwadratowego liczby ujemnej oraz wynik innych dziwnych operacji. Na przykład instrukcje `System.out.println(0.0/0.0)`; oraz `System.out.println(Math.sqrt(-1.0))`; zwracają wynik `NaN`.
- +0.0 jest zwracana jako wynik dzielenia liczby całkowitej przez +nieskończoność. Na przykład instrukcja `System.out.println(1.0/(1.0/0.0))`; zwróci wartość +0.0.
- -0.0 jest zwracana jako wynik dzielenia liczby ujemnej przez +nieskończoność. Na przykład instrukcja `System.out.println(-1.0/(1.0/0.0))`; zwróci wartość -0.0.

Jeżeli wynikiem jakiegoś działania jest wartość specjalna +nieskończoność, -nieskończoność lub NaN, wówczas całe wyrażenie ma zwykle taki sam wynik równy wartości specjalnej. Na przykład wynikiem instrukcji `System.out.println(1.0/0.0*20.0)`; jest `Infinity`. Jednak wyrażenie, w którym pojawia się wartość +nieskończoność lub -nieskończoność, może też zwrócić wartość NaN. Na przykład w wyrażeniu `1.0/0.0*0.0` najpierw pojawia się +nieskończoność (jako wynik wyrażenia `1.0/0.0`), a następnie NaN (jako wynik wyrażenia +nieskończoność\*0, 0).

Kolejna ciekawostka wiąże się z wartościami `Integer.MAX_VALUE`, `Integer.MIN_VALUE`, `Long.MAX_VALUE` oraz `Long.MIN_VALUE`. Każda z tych wartości jest klasą opakowującą typu podstawowego, która identyfikuje wartość maksymalną lub minimalną, jaka może być reprezentowana przez typ podstawowy skojarzony z klasą.

Można się także zastanawiać, dlaczego nie istnieją przeciążone wersje metod `abs()`, `max()` i `min()`, które obsługują argumenty typu `byte` i `short`, czyli `abs(byte b)` oraz `abs(short s)`. Wersje takie nie są jednak potrzebne, ponieważ ograniczone przedziały liczb całkowitych typu `byte` i `short` sprawiają, że są one mało przydatne do wykonywania obliczeń. Jeżeli jednak metody takie są potrzebne, można skorzystać z implementacji przedstawionej na listingu 6.2.

**Listing 6.2.** *Przeciążone metody `byte abs(byte b)` i `short abs(short s)`*

```
public static byte abs(byte b)
{
    return (b < 0) ? (byte) -b : b;
}
public static short abs(short s)
{
    return (s < 0) ? (short) -s : s;
}
public static void main(String[] args)
{
    byte b = -2;
    System.out.println(abs(b)); // Wynik: 2
    short s = -3;
    System.out.println(abs(s)); // Wynik: 3
}
```

Rzutowania (`byte`) i (`short`) trzeba wykonać, ponieważ wyrażenie `-b` przekształca wartość zmiennej `b` z typu `byte` na typ `int`, a wyrażenie `-s` przekształca wartość zmiennej `s` z typu `short` na typ `int`. Z kolei rzutowania nie są potrzebne w wyrażeniach `(b < 0)` i `(s < 0)`, ponieważ w tych przypadkach wartości `b` i `s` są automatycznie rzutowane do typu `int` przed porównaniem ich z wartością 0 typu `int`.

- 
- **Wskazówka** • Brak wspomnianych wersji metod w klasie `Math` mógłby sugerować, że typy `byte` i `short` nie są zbyt przydatne w deklaracjach metod. Jednak typy te przydają się wówczas, gdy deklaruje się tablice, których elementy przechowują małe wartości (na przykład wartości bajtowe w pliku binarnym). Gdyby tablica do przechowywania takich wartości została zadeklarowana jako typu `int` lub `long`, zmarnowałoby się w ten sposób znaczną ilość miejsca na stercie (a w skrajnym przypadku mogłoby to doprowadzić do wyczerpania się pamięci).
-

Gdy w dokumentacji języka Java analizuje się informacje na temat pakietu `java.lang`, można natknąć się na klasę o nazwie `StrictMath`. Oprócz dłuższej nazwy, klasa wydaje się identyczna z klasą `Math`. Różnice między dwiema klasami można podsumować następująco:

- Metody klasy `StrictMath` zwracają identyczne wyniki na wszystkich platformach. Natomiast metody klasy `Math` mogą zwracać nieco odmienne wartości, zależnie od platformy.
- Ponieważ `StrictMath` nie może używać elementów charakterystycznych dla konkretnych platform, takich jak choćby koprocesor obliczeń matematycznych o zwiększonej precyzji, implementacja klasy `StrictMath` może być mniej wydajna od implementacji klasy `Math`.

W większości przypadków metody klasy `Math` mogą wywoływać swoje odpowiedniczki z klasy `StrictMath`. Dwoma wyjątkami od tej reguły są metody `toDegrees()` i `toRadians()`. Wprawdzie w obydwóch klasach obie metody mają taką samą implementację, lecz w nagłówkach tych metod w klasie `StrictMath` występuje zastrzeżone słowo `strictfp`:

```
public static strictfp double toDegrees(double angrad)
public static strictfp double toRadians(double angdeg)
```

Według Wikipedii (<http://en.wikipedia.org/wiki/Strictfp>) słowo zastrzeżone `strictfp` ogranicza obliczenia zmiennopozycyjne w taki sposób, aby zapewnić przenośność. Słowo umożliwia przenośność dzięki zapewnieniu jednolitej pośredniej reprezentacji liczb zmiennopozycyjnych oraz w zakresie nadmiarów i niedomiarów (czyli generowania wartości zbyt dużej lub zbyt małej w stosunku do ograniczeń reprezentacji).

---

■ **Uwaga** • Zgodnie ze wspomnianym przed chwilą artykułem z Wikipedii na temat słowa zastrzeżonego `strictfp` klasa `Math` zawiera metodę `public static strictfp double abs(double)`; oraz inne metody `strictfp`. Jednak gdy w Javie 6 update 16 przeanalizuje się kod źródłowy tej klasy, okaże się, że nie ma w nim żadnego wystąpienia słowa `strictfp`. Jednak wiele metod klasy `Math` (na przykład metoda `sin()`) wywołuje swoje odpowiedniczki z klasy `StrictMath`, które są już zaimplementowane w bibliotece dla odpowiedniej platformy, a implementacje metod w tej bibliotece bazują już na `strictfp`.

---

Jeżeli słowo zastrzeżone `strictfp` nie jest obecne, wówczas obliczenia pośrednie nie są ograniczone do 32-bitowych i 64-bitowych reprezentacji zmiennopozycyjnych obsługiwanych przez Javę. W zamian w obliczeniach można korzystać z szerszych reprezentacji (w szczególności 128-bitowych) na tych platformach, które takie reprezentacje obsługują.

W przypadku reprezentacji wartości na 32 lub 64 bitach w trakcie obliczeń pośrednich może dojść do nadmiaru lub niedomiaru. Natomiast jeśli reprezentacja bazuje na większej liczbie bitów, prawdopodobieństwo wystąpienia nadmiaru lub niedomiaru się zmniejsza.

Ze względu na te różnice zapewnienie pełnej przenośności nie jest możliwe. Dlatego słowo zastrzeżone `strictfp` do pewnego stopnia wyrównuje te niespójności przez nałożenie wymagania, by na wszystkich platformach obliczenia pośrednie były wykonywane na reprezentacji 32-bitowej lub 64-bitowej.

Gdy w deklaracji metody zostanie zawarte słowo zastrzeżone `strictfp`, będzie ono gwarantować, że wszystkie wykonywane w metodzie obliczenia zmiennopozycyjne zostaną wykonane zgodnie z regułami zapewnienia precyzji. Słowo zastrzeżone `strictfp` można umieścić

również w deklaracji klasy (na przykład `public strictfp class FourierTransform`), aby zapewnić, że wszystkie obliczenia zmiennopozycyjne wykonywane w ramach tej klasy będą przeprowadzane z zapewnieniem właściwej precyzji.

- 
- **Uwaga** • Klasy `Math` i `StrictMath` są zadeklarowane jako `final` i dlatego nie można ich rozszerzać. Ponadto w klasach tych znajdują się deklaracje prywatnych, pustych i bezargumentowych konstruktorów, co powoduje, że nie można tworzyć ich instancji.

Klasy `Math` i `StrictMath` to przykłady tak zwanych **klas narzędziowych** (ang. *utility classes*), ponieważ wyznaczają one obszar zarezerwowany dla stałych narzędziowych oraz metod narzędziowych (`static`).

---

## Klasa `BigDecimal`

W rozdziale 2. zdefiniowano klasę `CheckingAccount` z polem o nazwie `balance`. Zgodnie z deklaracją pole `balance` jest typu `int`, a dodatkowo w kodzie źródłowym został umieszczony komentarz, według którego pole `balance` reprezentuje liczbę złotych, jaką można wypłacić z konta. Alternatywnie można było wskazać, że pole `balance` zawiera liczbę groszy dostępnych do wypłaty.

Można zadać pytanie, dlaczego pole `balance` nie zostało zadeklarowane jako pole typu `double` lub `float`. Dzięki temu w polu `balance` można by przechowywać na przykład wartość 18,26 (18 złotych w części całkowitoliczbowej i 26 groszy w części ułamkowej). Pole `balance` nie zostało zadeklarowane jako typu `float` lub `double` z następujących względów:

- Nie wszystkie wartości zmiennopozycyjne, które mogą reprezentować kwoty pieniężne (w złotych i groszach), mogą być przechowywane w pamięci z odpowiednią precyzją. Na przykład liczba 0.1 (która może oznaczać 10 groszy) nie posiada precyzyjnej reprezentacji w pamięci. Gdyby wykonano wyrażenie `double total = 0.1; for (int i = 0; i < 50; i++) total += 0.1; System.out.println(total);`, jego wynikiem byłaby wartość 5.099999999999998, a nie prawidłowa wartość 5.1.
- Wynik każdego obliczenia zmiennopozycyjnego musi zostać zaokrąglony do jednego grosza. Jeżeli tak się nie stanie, w obliczeniach pojawią się drobne niedokładności, które w efekcie mogą spowodować, że ostateczny wynik będzie się znacznie różnił od prawidłowego. Wprawdzie klasa `Math` udostępnia dwie metody `round()`, za pomocą których można próbować zaokrąglać wyniki obliczeń do jednego grosza, lecz metody te zaokrąglają wartości do najbliższych liczb całkowitych (czyli złotych).

Aplikacja `InvoiceCalc` z listingu 6.3 ilustruje obydwa wspomniane problemy. Jednak pierwszy z tych problemów nie jest aż tak poważny, ponieważ nie wpływa znacząco na dokładność ostatecznego wyniku. Zdecydowanie bardziej istotny jest drugi problem, przez który wyniki obliczeń nie są zaokrąglane do jednego grosza.

**Listing 6.3.** Obliczanie zmiennopozycyjnej wartości faktury, które prowadzi do powstania błędów

```
import java.text.NumberFormat;

class InvoiceCalc
{
```



```

final static double DISCOUNT_PERCENT = 0.1; // 10%
final static double TAX_PERCENT = 0.05; // 5%
public static void main(String[] args)
{
    double invoiceSubtotal = 285.36;
    double discount = invoiceSubtotal*DISCOUNT_PERCENT;
    double subtotalBeforeTax = invoiceSubtotal-discount;
    double salesTax = subtotalBeforeTax*TAX_PERCENT;
    double invoiceTotal = subtotalBeforeTax+salesTax;
    NumberFormat currencyFormat = NumberFormat.getCurrencyInstance();
    System.out.println("Suma: " + currencyFormat.format(invoiceSubtotal));
    System.out.println("Rabat: " + currencyFormat.format(discount));
    System.out.println("Suma po uwzględnieniu rabatu: " +
        currencyFormat.format(subtotalBeforeTax));
    System.out.println("Podatek: " + currencyFormat.format(salesTax));
    System.out.println("Łącznie: " + currencyFormat.format(invoiceTotal));
}
}

```

Na listingu 6.3 wykorzystano klasę `NumberFormat` (z pakietu `java.text`) i jej metodę `format()`, aby sformatować wartość zmiennopozycyjną o podwójnej precyzji na postać wartości walutowej. Więcej na temat klasy `NumberFormat` powiemy w rozdziale 9. Gdy uruchomi się aplikacja `InvoiceCalc`, zwróci ona następujący wynik:

```

Suma: 285,36 zł
Rabat: 28,54 zł
Suma po uwzględnieniu rabatu: 256,82 zł
Podatek: 12,84 zł
łącznie: 269,67 zł

```

W wynikach działania aplikacji prawidłowo obliczone zostały kwoty sumy, rabatu, sumy po uwzględnieniu rabatu i podatku. Nieprawidłowo natomiast wskazana jest kwota łączna, której wartość wynosi 269,67 zamiast 269,66. Klient nie będzie skłonny zapłacić dodatkowego grosza, mimo że zgodnie z regułami obliczeń zmiennopozycyjnych to właśnie kwota 269,67 jest wartością prawidłową:

```

Suma: 285,36
Rabat: 28,536
Suma po uwzględnieniu rabatu: 256,824
Podatek: 12,8412
łącznie: 269,6652

```

Źródłem problemu jest to, że po wykonaniu każdej operacji, a przed wykonaniem operacji następnej uzyskany wynik nie jest zaokrąglany do najbliższego grosza. W efekcie 0,024 w liczbie 256,824 oraz 0,012 w liczbie 12,84 są uwzględniane także w kwocie końcowej, przez co metoda `format()` klasy `NumberFormat` zaokrągliła kwotę końcową do wartości 269,67.

Java udostępnia rozwiązanie obydwóch problemów w postaci klasy `java.math.BigDecimal`. Jest to klasa niezmienna (to znaczy instancja klasy `BigDecimal` nie może być zmieniona), która reprezentuje liczbę dziesiętną o określonym znaku (na przykład 23,653) ze wskazaną **precyzją** (liczbą cyfr) i odpowiednią **skalą** (czyli z uwzględnieniem liczby całkowitej wyznaczającej liczbę cyfr po przecinku).

W klasie `BigDecimal` zadeklarowano trzy wygodne stałe: `ONE`, `TEN` i `ZERO`. Każda z tych stałych jest odpowiednikiem wartości 1, 10 i 0 ze skalą zerową.

---

■ **Ostrzeżenie** • W klasie `BigDecimal` zadeklarowano kilka stałych, których nazwy zaczynają się od słowa `ROUND_`. Stałe te są w większości przestarzałe i powinno się ich unikać. To samo dotyczy metod `public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode)` oraz `public BigDecimal setScale(int newScale, int roundingMode)`, które są nadal obecne w klasie, aby zagwarantować prawidłową kompilację starszego kodu.

---

Klasa `BigDecimal` deklaruje ponadto kilka przydatnych konstruktorów i metod. Wybrane konstruktory i metody zostały opisane w tabeli 6.2.

**Tabela 6.2.** *Konstruktory i metody klasy `BigDecimal`*

Metoda	Opis
<code>BigDecimal(int val)</code>	Inicjalizuje instancję klasy <code>BigDecimal</code> liczbą cyfr wskazywaną przez <code>val</code> i skalą 0.
<code>BigDecimal(String val)</code>	Inicjalizuje instancję klasy <code>BigDecimal</code> dziesiętnym odpowiednikiem <code>val</code> . Jako skalę ustawia liczbę cyfr po przecinku lub 0, jeżeli przecinek nie występuje. Jeśli <code>val</code> będzie <code>null</code> , konstruktor rzuci wyjątek <code>java.lang.NullPointerException</code> . Jeżeli reprezentacja <code>val</code> w postaci ciągu znaków będzie nieprawidłowa (na przykład będzie zawierała litery), konstruktor rzuci wyjątek <code>java.lang.NumberFormatException</code> .
<code>BigDecimal abs()</code>	Zwraca nową instancję klasy <code>BigDecimal</code> , która zawiera bezwzględną wartość wartości instancji bieżącej. Skala nowej instancji jest taka sama jak skala instancji bieżącej.
<code>BigDecimal add(BigDecimal augend)</code>	Zwraca nową instancję klasy <code>BigDecimal</code> , która zawiera sumę wartości bieżącej oraz argumentu przekazanego do konstruktora. Skala w nowej instancji jest wyznaczana przez wartość większą spośród skali instancji bieżącej i skali instancji przekazanej jako argument. Jeżeli <code>augend</code> będzie <code>null</code> , metoda rzuci wyjątek <code>NullPointerException</code> .
<code>BigDecimal divide(BigDecimal divisor)</code>	Zwraca nową instancję klasy <code>BigDecimal</code> , która zawiera iloraz wartości bieżącej podzielonej i wartości argumentu. Skala nowej instancji to różnica między skalą instancji bieżącej i skalą instancji przekazanej jako argument. Skala ta może zostać odpowiednio dostosowana, jeżeli do przedstawienia wyniku dzielenia potrzebna będzie większa liczba cyfr. Jeżeli <code>divisor</code> będzie <code>null</code> , metoda rzuci wyjątek <code>NullPointerException</code> , a jeżeli <code>divisor</code> będzie reprezentować wartość 0 lub wyniku dzielenia nie będzie można zaprezentować precyzyjnie, metoda rzuci wyjątek <code>java.lang.ArithmeticException</code> .
<code>BigDecimal max(BigDecimal val)</code>	Zwraca <code>this</code> lub <code>val</code> , zależnie od tego, która z tych instancji posiada większą wartość. Jeżeli <code>val</code> będzie <code>null</code> , metoda rzuci wyjątek <code>NullPointerException</code> .
<code>BigDecimal min(BigDecimal val)</code>	Zwraca <code>this</code> lub <code>val</code> , zależnie od tego, która z tych instancji posiada mniejszą wartość. Jeżeli <code>val</code> będzie <code>null</code> , metoda rzuci wyjątek <code>NullPointerException</code> .

**Tabela 6.2.** Konstruktory i metody klasy `BigDecimal` — ciąg dalszy

Metoda	Opis
<code>BigDecimal multiply(BigDecimal multiplicand)</code>	Zwraca nową instancję klasy <code>BigDecimal</code> , która zawiera iloczyn wartości bieżącej i wartości instancji podanej jako argument. Skala nowej instancji jest sumą skali instancji bieżącej i instancji przekazanej jako argument. Jeżeli <code>multiplicand</code> jest <code>null</code> , metoda rzuci wyjątek <code>NullPointerException</code> .
<code>BigDecimal negate()</code>	Zwraca nową instancję klasy <code>BigDecimal</code> , która zawiera wartość przeciwną do wartości bieżącej. Skala nowej instancji jest taka sama jak skala instancji bieżącej.
<code>int precision()</code>	Zwraca precyzję bieżącej instancji klasy <code>BigDecimal</code> .
<code>BigDecimal remainder(BigDecimal divisor)</code>	Zwraca nową instancję klasy <code>BigDecimal</code> , która zawiera resztę z dzielenia wartości instancji bieżącej przez wartość instancji przekazanej jako argument. Skala nowej instancji to różnica między skalą bieżącą a skalą argumentu. Skala ta może zostać odpowiednio dostosowana, jeżeli do wyświetlenia wyniku potrzeba będzie większej liczby cyfr. Jeżeli <code>divisor</code> będzie <code>null</code> , metoda rzuci wyjątek <code>NullPointerException</code> , a jeśli <code>divisor</code> będzie mieć wartość 0, metoda rzuci wyjątek <code>ArithmeticException</code> .
<code>int scale()</code>	Zwraca skalę bieżącej instancji klasy <code>BigDecimal</code> .
<code>BigDecimal setScale(int newScale, RoundingMode roundingMode)</code>	Zwraca nową instancję klasy <code>BigDecimal</code> o wskazanej skali i trybie zaokrąglania. Jeżeli nowa skala jest większa niż skala dotychczasowa, wówczas do wartości, które były niewyskalowane, dodawane są zera. W takiej sytuacji nie potrzeba wykonywać zaokrąglania. Jeżeli nowa skala jest mniejsza niż skala dotychczasowa, wówczas usuwane są ostatnie cyfry. Jeżeli usuwane cyfry są inne niż zero, uzyskana w ten sposób liczba niewyskalowana musi zostać zaokrąglona. Zaokrąglenie wykonuje się w trybie zaokrąglania wskazanym jako argument. Jeżeli <code>roundingMode</code> jest <code>null</code> , metoda rzuci wyjątek <code>NullPointerException</code> . Jeżeli natomiast <code>roundingMode</code> będzie mieć wartość <code>RoundingMode.ROUND_UNNECESSARY</code> , a ze względu na bieżącą skalę zaokrąglenie będzie potrzebne, metoda rzuci wyjątek <code>ArithmeticException</code> .
<code>BigDecimal subtract(BigDecimal subtrahend)</code>	Zwraca nową instancję klasy <code>BigDecimal</code> , która zawiera wartość bieżącą pomniejszoną o wartość argumentu. Skala nowej instancji jest większą spośród skali bieżącej i skali argumentu. Jeżeli <code>subtrahend</code> będzie <code>null</code> , metoda rzuci wyjątek <code>NullPointerException</code> .
<code>String toString()</code>	Zwraca ciąg znaków, który stanowi reprezentację bieżącej instancji <code>BigDecimal</code> . Jeżeli to konieczne, zostanie użyta notacja naukowa.

W tabeli 6.2 wspomniano o typie `RoundingMode` wyznaczającym tryb zaokrąglania. `RoundingMode` to typ wyliczeniowy enum, który zawiera stałe odpowiadające różnorodnym trybom zaokrąglania. Stałe te opisano w tabeli 6.3.

Najlepszym sposobem oswojenia się z klasą `BigDecimal` jest zastosowanie jej w praktyce. Na listingu 6.4 wykorzystano tę klasę do wykonania prawidłowych obliczeń wartości na fakturze, którą przedstawiono wcześniej na listingu 6.3.

**Tabela 6.3.** Stałe typu *RoundingMode*

Stała	Opis
CEILING	Zaokrąga w kierunku plus nieskończoności.
DOWN	Zaokrąga w kierunku zera.
FLOOR	Zaokrąga w kierunku minus nieskończoności.
HALF_DOWN	Zaokrąga w kierunku „bliźszej liczby sąsiedniej”, chyba że odległość do liczb sąsiednich jest taka sama — wówczas zaokrąga w dół.
HALF_EVEN	Zaokrąga w kierunku „bliźszej liczby sąsiedniej”, chyba że odległość do liczb sąsiednich jest taka sama — wówczas zaokrąga w stronę tej liczby sąsiedniej, która jest parzysta.
HALF_UP	Zaokrąga w kierunku „bliźszej liczby sąsiedniej”, chyba że odległość do liczb sąsiednich jest taka sama — wówczas zaokrąga w górę (o tym trybie zaokrąglania najczęściej uczy się w szkole).
UNNECESSARY	Zaokrąglanie nie jest potrzebne, ponieważ wynik działania jest zawsze dokładny.
UP	Liczby dodatnie są zaokrąglane w kierunku plus nieskończoności, a liczby ujemne są zaokrąglane w kierunku minus nieskończoności.

**Listing 6.4.** Obliczenia wartości na fakturze przy użyciu klasy *BigDecimal*, dzięki której unika się błędów zaokrągleń

```
class InvoiceCalc
{
    public static void main(String[] args)
    {
        BigDecimal invoiceSubtotal = new BigDecimal("285.36");
        BigDecimal discountPercent = new BigDecimal("0.10");
        BigDecimal discount = invoiceSubtotal.multiply(discountPercent);
        discount = discount.setScale(2, RoundingMode.HALF_UP);
        BigDecimal subtotalBeforeTax = invoiceSubtotal.subtract(discount);
        subtotalBeforeTax = subtotalBeforeTax.setScale(2, RoundingMode.HALF_UP);
        BigDecimal salesTaxPercent = new BigDecimal("0.05");
        BigDecimal salesTax = subtotalBeforeTax.multiply(salesTaxPercent);
        salesTax = salesTax.setScale(2, RoundingMode.HALF_UP);
        BigDecimal invoiceTotal = subtotalBeforeTax.add(salesTax);
        invoiceTotal = invoiceTotal.setScale(2, RoundingMode.HALF_UP);
        System.out.println("Suma: " + invoiceSubtotal);
        System.out.println("Rabat: " + discount);
        System.out.println("Suma po uwzględnieniu rabatu: " + subtotalBeforeTax);
        System.out.println("Podatek: " + salesTax);
        System.out.println("Łącznie: " + invoiceTotal);
    }
}
```

W metodzie `main()` z listingu 6.4 najpierw tworzone są obiekty `BigDecimal` o nazwach `invoiceSubtotal` i `discountPercent`, inicjalizowane wartościami odpowiednio 285.36 i 0.10. Wartość `invoiceSubtotal` jest mnożona przez `discountPercent`, a wynikowa instancja klasy `BigDecimal` tego mnożenia jest przypisywana zmiennej `discount`.

Na tym etapie zmienna `discount` ma wartość 28.5360. Jeśli nie liczyć ostatniego zera, wartość ta jest taka sama jak wartość wyrażenia `invoiceSubtotal*DISCOUNT_PERCENT`

z listingu 6.3. Wartością, która powinna zostać przypisana zmiennej `discount`, jest 28.54. Aby rozwiązać ten problem jeszcze przed wykonaniem kolejnego obliczenia, w metodzie `main()` jest wywoływana metoda `setScale()` z następującymi argumentami:

- 2 — dwie cyfry po przecinku.
- `RoundMode.HALF_UP` — standardowy sposób zaokrąglania.

Gdy określona jest już odpowiednia skala i tryb zaokrąglania, metoda `main()` odejmuje `discount` od `invoiceSubtotal`, a instancję `BigDecimal` stanowiącą reprezentację uzyskanej różnicy przypisuje zmiennej `subtotalBeforeTax`. Następnie na zmiennej `subtotalBeforeTax` jest wywoływana metoda `setScale()`, aby odpowiednio zaokrąglić jej wartość przed wykonaniem kolejnego obliczenia.

W kolejnym kroku `main()` tworzy obiekt klasy `BigDecimal` o nazwie `salesTaxPercent`, który zostaje zainicjalizowany wartością 0.05. Dalej `subtotalBeforeTax` zostaje przemnożona przez `salesTaxPercent`, a iloczyn jest przypisywany zmiennej `salesTax`. Na tej zmiennej, której typem jest `BigDecimal`, jest wywoływana metoda `setScale()`, aby odpowiednio zaokrąglić wartość reprezentowaną przez ten obiekt.

Po wykonaniu tej czynności `main()` dodaje `salesTax` do `subtotalBeforeTax`. Suma zostaje przypisana zmiennej `invoiceTotal`, na której metoda `setScale()` dokonuje odpowiedniego zaokrąglania. Na koniec wartości wszystkich obiektów klasy `BigDecimal` zostają przekazane do standardowego wyjścia za pomocą metody `System.out.println()`, która wywołuje na każdym obiekcie ich metody `toString()`, aby uzyskać ciąg znaków reprezentujący wartości poszczególnych instancji `BigDecimal`.

Gdy tak zmodyfikowana wersja aplikacji `InvoiceCalc` zostanie uruchomiona, uzyskamy następujący wynik:

```
Suma: 285,36
Rabat: 28,54
Suma po uwzględnieniu rabatu: 256,82
Podatek: 12,84
Łącznie: 269,66
```

- 
- **Ostrzeżenie** • Klasa `BigDecimal` deklaruje konstruktor `BigDecimal(double val)`, którego w miarę możliwości powinno się unikać. Konstruktor ten inicjalizuje instancję klasy `BigDecimal` wartością przekazaną jako `val`, przez co w sytuacji gdy wartość typu `double` nie może zostać przedstawiona jako wartość dokładna, tak utworzona instancja może odzwierciedlać nieprawidłową reprezentację wartości. Na przykład wynikiem wywołania `BigDecimal(0.1)` jest instancja z wartością 0.10000000000000000555111512↵31257827021181583404541015625. Z kolei wywołanie `BigDecimal("0.1")` daje w wyniku instancję z prawidłową wartością 0.1.
- 

## Klasa `BigInteger`

Klasa `BigDecimal` przechowuje liczbę dziesiętną ze znakiem w postaci niewyskalowanej wartości ze skalą określoną przez 32-bitową liczbę całkowitą. Wartość niewyskalowana jest przechowywana w instancji klasy `java.math.BigInteger`.

Klasa `BigInteger` jest klasą niezmienną, która reprezentuje liczbę całkowitą ze znakiem o określonej precyzji. Wartość jest w tej klasie przechowywana w **kodzie uzupełnień do dwóch** (inaczej mówiąc, wszystkie bity są zamienione — jedynki na zera, a zera na jedynki — a do wyniku dodawana jest wartość 1, aby wynik był zgodny z kodem uzupełnień do dwóch używanym w typach języka Java: całkowitoliczbowym bajtowym, krótkim całkowitoliczbowym, całkowitoliczbowym i długim całkowitoliczbowym).

---

■ **Uwaga** • Więcej informacji na temat zapisu w kodzie uzupełnień do dwóch można znaleźć w Wikipedii, na stronie pod adresem [http://pl.wikipedia.org/wiki/Kod\\_uzupe%C5%82nie%C5%84\\_do\\_dw%C3%B3ch](http://pl.wikipedia.org/wiki/Kod_uzupe%C5%82nie%C5%84_do_dw%C3%B3ch).

---

W klasie `BigInteger` są zadeklarowane trzy przydatne stałe o nazwach `ONE`, `TEN` i `ZERO`. Każda z tych stałych odpowiada instancji klasy `BigInteger` reprezentującej wartości odpowiednio 1, 10 i 0.

W klasie `BigInteger` jest zadeklarowana całkiem pokaźna liczba przydatnych konstruktorów i metod. Kilka wybranych konstruktorów i metod przedstawiono w tabeli 6.4.

Najlepszym sposobem, aby ośwoić się z klasą `BigInteger`, jest wypróbowanie jej w praktyce. Na listingu 6.5 wykorzystano klasę `BigInteger` do zaimplementowania metod `factorial`, aby porównać sposób ich działania.

**Listing 6.5.** *Porównanie metod `factorial()`*

```
class FactComp
{
    public static void main(String[] args)
    {
        System.out.println(factorial(12));
        System.out.println();
        System.out.println(factorial(20L));
        System.out.println();
        System.out.println(factorial(170.0));
        System.out.println();
        System.out.println(factorial(new BigInteger("170")));
        System.out.println();
        System.out.println(factorial(25.0));
        System.out.println();
        System.out.println(factorial(new BigInteger("25")));
    }
    public static int factorial(int n)
    {
        if (n == 0)
            return 1;
        else
            return n*factorial(n-1);
    }
    public static long factorial(long n)
    {
        if (n == 0)
            return 1;
        else
            return n*factorial(n-1);
    }
    public static double factorial(double n)
```

Tabela 6.4. Konstruktory i metody klasy *BigInteger*

Metoda	Opis
<code>BigInteger(byte[] val)</code>	Inicjalizuje instancję klasy <code>BigInteger</code> liczbą całkowitą przechowywaną w tablicy <code>val</code> . Element <code>val[0]</code> tej tablicy zawiera osiem najbardziej znaczących (lewych) bitów. Jeżeli <code>val</code> będzie <code>null</code> , konstruktor rzuci wyjątek <code>NullPointerException</code> , natomiast jeśli <code>val.length</code> będzie mieć wartość 0, konstruktor rzuci wyjątek <code>NumberFormatException</code> .
<code>BigInteger(String val)</code>	Inicjalizuje instancję klasy <code>BigInteger</code> liczbą całkowitą odpowiadającą <code>val</code> . Jeżeli <code>val</code> będzie <code>null</code> , konstruktor rzuci wyjątek <code>NullPointerException</code> , a jeśli ciąg znaków reprezentujący <code>val</code> będzie nieprawidłowy (na przykład będzie zawierać litery), wówczas konstruktor rzuci wyjątek <code>NumberFormatException</code> .
<code>BigInteger abs()</code>	Zwraca nową instancję klasy <code>BigInteger</code> , która zawiera wartość bezwzględną wartości instancji bieżącej.
<code>BigInteger add(BigInteger augend)</code>	Zwraca nową instancję klasy <code>BigInteger</code> , która zawiera sumę wartości bieżącej oraz wartości przekazanej jako argument. Jeżeli <code>augend</code> będzie <code>null</code> , metoda rzuci wyjątek <code>NullPointerException</code> .
<code>BigInteger divide(BigInteger divisor)</code>	Zwraca nową instancję klasy <code>BigInteger</code> , która zawiera ilorz wartości bieżącej podzielonej przez wartość przekazaną jako argument. Jeżeli <code>divisor</code> jest <code>null</code> , metoda rzuci <code>NullPointerException</code> , natomiast jeśli <code>divisor</code> reprezentuje wartość 0 lub wynik dzielenia nie jest wartością dokładną, metoda rzuci wyjątek <code>ArithmeticException</code> .
<code>BigInteger max(BigInteger val)</code>	Zwraca <code>this</code> lub <code>val</code> zależnie od tego, która instancja klasy <code>BigInteger</code> zawiera większą wartość. Jeżeli <code>val</code> jest <code>null</code> , metoda rzuci wyjątek <code>NullPointerException</code> .
<code>BigInteger min(BigInteger val)</code>	Zwraca <code>this</code> lub <code>val</code> zależnie od tego, która instancja klasy <code>BigInteger</code> zawiera mniejszą wartość. Jeżeli <code>val</code> jest <code>null</code> , metoda rzuci wyjątek <code>NullPointerException</code> .
<code>BigInteger multiply(BigInteger multiplicand)</code>	Zwraca nową instancję klasy <code>BigInteger</code> , która zawiera iloczyn wartości bieżącej oraz wartości argumentu. Jeżeli <code>multiplicand</code> będzie <code>null</code> , metoda rzuci <code>NullPointerException</code> .
<code>BigInteger negate()</code>	Zwraca nową instancję klasy <code>BigInteger</code> , która zawiera wartość przeciwną do wartości bieżącej.
<code>BigInteger remainder(BigInteger divisor)</code>	Zwraca nową instancję klasy <code>BigInteger</code> , która zawiera resztę z podzielenia wartości bieżącej przez wartość argumentu. Jeżeli <code>divisor</code> jest <code>null</code> , metoda rzuci wyjątek <code>NullPointerException</code> , natomiast jeżeli <code>divisor</code> reprezentuje wartość 0, metoda rzuci wyjątek <code>ArithmeticException</code> .
<code>BigInteger subtract(BigInteger subtrahend)</code>	Zwraca nową instancję klasy <code>BigInteger</code> , która zawiera wartość bieżącą pomniejszoną o wartość argumentu. Jeżeli <code>subtrahend</code> będzie <code>null</code> , metoda rzuci wyjątek <code>NullPointerException</code> .
<code>String toString()</code>	Zwraca ciąg znaków stanowiący reprezentację instancji <code>BigInteger</code> .

```

{
    if (n == 1.0)
        return 1.0;
    else
        return n*factorial(n-1);
}
public static BigInteger factorial(BigInteger n)
{
    if (n.equals(BigInteger.ZERO))
        return BigInteger.ONE;
    else
        return n.multiply(factorial(n.subtract(BigInteger.ONE)));
}
}

```

Na listingu 6.5 porównano cztery wersje rekurencyjnej metody `factorial()`, której zadaniem jest wyliczenie silni. Porównanie to pozwala zidentyfikować największą wartość argumentu, jaką można przekazać do pierwszych trzech metod, zanim metody te zaczną zwracać nieprawidłowe wartości spowodowane przekroczeniem przedziału wartości poprawnie reprezentowanych przez dany typ liczbowy.

W pierwszej wersji metody wykorzystywany jest argument typu `int`, a przedział wartości przekazywanego argumentu wynosi od 0 do 12. Jeżeli do metody zostanie przekazany dowolny argument większy od 12, wartości silni tego argumentu nie będzie już można zaprezentować prawidłowo jako liczby typu `int`.

Przedział, do jakiego może należeć argument metody `factorial()`, można poszerzyć przez zmianę typu parametru na `long`. Przedział ten nie zwiększy się jednak znacząco. Po wprowadzeniu zmiany typu argumentu do metody będzie można przekazać wartość nie większą niż 20.

Aby jeszcze bardziej zwiększyć zakres obsługiwanych wartości, można zaimplementować metodę `factorial()`, której parametr i zwracana wartość będą typu `double`. Jest to możliwe, ponieważ za pomocą typu `double` można precyzyjnie reprezentować liczby całkowite. Jednak w takim przypadku największym argumentem, jaki można przekazać do metody, jest 170.0. Każda wartość większa niż 170.0 spowoduje, że metoda `factorial()` zwróci plus nieskończoność.

Może się zdarzyć sytuacja, że konieczne będzie wyliczenie silni wartości większej niż 170 — na przykład gdy będą wykonywane pewne obliczenia statystyczne uwzględniające kombinacje lub permutacje. Jedyny sposób, który pozwoli na obliczanie takiej silni, polega na użyciu wersji metody `factorial()` wykorzystującej typ `BigInteger`.

Gdy aplikacja z listingu 6.5 zostanie uruchomiona, zwróci następujące wyniki:

479001600

2432902008176640000

7.257415615307994E306

72574156153079989673967282111292631147169916812964513765435777989005618434017061578523507492  
42617459511490991237838520776666022565442753025328900773207510902400430280058295603966612599  
65825710439855829425756896631343961226257109494680671120556888045719334021266145280000000000  
00

1.5511210043330986E25

15511210043330985984000000



Pierwsze trzy wartości to najwyższe wartości silni zwracane przez metody `factorial()`, w których wykorzystano odpowiednio typy `int`, `long` i `double`. Czwarta wartość reprezentuje wartość typu `BigInteger`, która odpowiada najwyższej wartości silni typu `double`.

Warto zauważyć, że metoda z typem `double` nie jest w stanie dokładnie zaprezentować wartości 170! (! to matematyczny symbol silni). Precyzja tej metody jest po prostu zbyt mała. Wprawdzie metoda próbuje zaokrąglić najmniejszą cyfrę, lecz takie zaokrąglenie nie zawsze się udaje — liczba kończy się cyframi 7994 zamiast 7998. Zaokrąglenie sprawdza się jedynie w przypadkach argumentów nie większych niż 25.0, o czym świadczą dwa ostatnie wiersze danych wyników.

- 
- **Uwaga** • Algorytm szifrowania RSA, klasa `BigDecimal` i obliczanie silni to praktyczne przykłady zastosowania klasy `BigInteger`. Klasy tej można używać jednak także w zastosowaniach niestandardowych. Na przykład w artykule opublikowanym w magazynie „JavaWorld” z lutego 2006 pod tytułem *Travel Through Time with Java* (<http://www.javaworld.com/javaworld/jw-02-2006/jw-0213-funandgames.html>), który stanowił kolejną część serii *Java Fun and Games*, została wykorzystana klasa `BigInteger` do przechowywania obrazka w postaci bardzo dużej liczby całkowitej. Pomysł polegał na tym, by poeksperymentować z klasą `BigInteger` i sprawdzić, jak za jej pomocą można wyszukiwać zdjęcia ludzi i miejsc, które istniały w przeszłości, będą istnieć w przyszłości lub nigdy nie istniały. Artykuł można polecić zwłaszcza programistom, którzy lubią trochę poszaleć.
- 

## Informacje na temat pakietów

Klasa `java.lang.Package` udostępnia informacje na temat pakietu (wstępne informacje na temat pakietów zostały przedstawione w rozdziale 4.). Informacje te dotyczą wersji implementacji i specyfikacji pakietu języka Java, nazwy pakietu, a także wskazują, czy pakiet został **upakowany** (ang. *sealed*), czyli czy wszystkie klasy wchodzące w skład pakietu zostały zarchiwizowane w tym samym pliku JAR.

- 
- **Uwaga** • Pliki JAR zostały opisane w rozdziale 1.
- 

W tabeli 6.5 opisano wybrane metody klasy `Package`.

Na listingu 6.6 znajduje się kod źródłowy aplikacji `PackageInfo`, którą zaimplementowano, aby zaprezentować działanie większości metod klasy `Package` z tabeli 6.5.

**Listing 6.6.** *Uzyskiwanie informacji na temat pakietu*

```
public class PackageInfo
{
    public static void main(String[] args)
    {
        if (args.length == 0)
        {
            System.err.println("użycie: java PackageInfo nazwaPakietu [wersja]");
            return;
        }
        Package pkg = Package.getPackage(args[0]);
        if (pkg == null)
```

**Tabela 6.5.** Metody klasy *Package*

Metoda	Opis
<code>String getImplementationTitle()</code>	Zwraca tytuł implementacji pakietu, który może być <code>null</code> . Format tytułu jest nieokreślony.
<code>String getImplementationVendor()</code>	Zwraca nazwę sprzedawcy lub organizacji, która dostarczyła implementację pakietu. Nazwa ta może być <code>null</code> . Format nazwy jest nieokreślony.
<code>String getImplementationVersion()</code>	Zwraca numer wersji implementacji pakietu, który może być <code>null</code> . Ciąg reprezentujący numer wersji musi być sekwencją dodatnich dziesiętnych liczb całkowitych, oddzielonych od siebie znakami kropki. Liczby te mogą mieć poprzedzające zera.
<code>String getName()</code>	Zwraca nazwę pakietu w standardowym zapisie z kropkami, na przykład <code>java.lang</code> .
<code>static Package getPackage(String packageName)</code>	Zwraca obiekt klasy <code>Package</code> , który jest skojarzony z pakietem wskazywanym przez <code>packageName</code> . Jeżeli pakietu wskazywanego przez <code>packageName</code> nie można znaleźć, zwraca <code>null</code> . Jeżeli <code>packageName</code> jest <code>null</code> , metoda zwraca <code>NullPointerException</code> .
<code>static Package[] getPackages()</code>	Zwraca tablicę wszystkich obiektów klasy <code>Package</code> , które są dostępne dla kodu wywołującego.
<code>String getSpecificationTitle()</code>	Zwraca tytuł specyfikacji pakietu, który może być <code>null</code> . Format tytułu jest nieokreślony.
<code>String getSpecificationVendor()</code>	Zwraca nazwę sprzedawcy lub organizacji, która dostarczyła specyfikację zaimplementowaną w ramach pakietu. Nazwa ta może być <code>null</code> . Format nazwy jest nieokreślony.
<code>String getSpecificationVersion()</code>	Zwraca numer wersji specyfikacji implementacji pakietu, który może być <code>null</code> . Ciąg wskazujący numer wersji musi być sekwencją dodatnich dziesiętnych liczb całkowitych oddzielonych od siebie znakami kropek. Liczby te mogą mieć poprzedzające zera.
<code>boolean isCompatibleWith(String desired)</code>	Sprawdza, czy pakiet jest zgodny z podanym ciągiem wersji. W tym celu wersja specyfikacji pakietu jest porównywana z wersją <code>desired</code> . Jeżeli numer wersji specyfikacji pakietu jest wyższy od pożądanego numeru wersji <code>desired</code> lub mu równy, metoda zwraca wartość <code>true</code> (co oznacza, że pakiet jest zgodny); w przeciwnym razie metoda zwraca wartość <code>false</code> . Jeżeli <code>desired</code> jest <code>null</code> , metoda rzuca wyjątek <code>NullPointerException</code> , a jeżeli numer wersji pakietu lub numer wersji <code>desired</code> nie ma formatu z kropkami, metoda rzuca wyjątek <code>NumberFormatException</code> .
<code>boolean isSealed()</code>	Zwraca wartość <code>true</code> , jeżeli pakiet jest upakowany. W przeciwnym razie zwracana jest wartość <code>false</code> .

```

    {
        System.err.println("Nie znaleziono pakietu " + args[0]);
        return;
    }
    System.out.println("Nazwa: " + pkg.getName());
    System.out.println("Tytuł implementacji: " +
        pkg.getImplementationTitle());
    System.out.println("Dostawca implementacji: " +
        pkg.getImplementationVendor());
    System.out.println("Wersja implementacji: " +
        pkg.getImplementationVersion());
    System.out.println("Tytuł specyfikacji: " +
        pkg.getSpecificationTitle());
    System.out.println("Dostawca specyfikacji: " +
        pkg.getSpecificationVendor());
    System.out.println("Wersja specyfikacji: " +
        pkg.getSpecificationVersion());
    System.out.println("Upakowany: " + pkg.isSealed());
    if (args.length > 1)
        System.out.println("Zgodny z wersją " + args[1] + ": " +
            pkg.isCompatibleWith(args[1]));
    }
}

```

Aby skorzystać z aplikacji, w wierszu poleceń należy podać przynajmniej nazwę pakietu. Na przykład polecenie `java PackageInfo java.lang` zwraca w wersji 6 języka Java następujące dane wynikowe:

```

Nazwa: java.lang
Tytuł implementacji: Java Runtime Environment
Dostawca implementacji: Sun Microsystems, Inc.
Wersja implementacji: 1.6.0_16
Tytuł specyfikacji: Java Platform API Specification
Dostawca specyfikacji: Sun Microsystems, Inc.
Wersja specyfikacji: 1.6
Upakowany: false

```

Aplikacja `PackageInfo` pozwala ustalić, czy specyfikacja pakietu jest zgodna z określonym numerem wersji. Pakiet jest zgodny ze swoimi poprzednikami.

Na przykład jeżeli aplikację wywoła się poleceniem `java PackageInfo java.lang 1.6`, zwróci ona wynik `Zgodny z wersją 1.6: true`, podczas gdy aplikacja wywołana poleceniem `java PackageInfo java.lang 1.8` zwróci wynik `Zgodny z wersją 1.6: false`.

Za pomocą aplikacji `PackageInfo` można także analizować pakiety utworzone samodzielnie. O tym, jak tworzy się własne pakiety, była już mowa w rozdziale 4. W przykładzie 4. zaprezentowano przykładowy pakiet `logging`.

Plik `PackageInfo.class` należy skopiować do katalogu, w którym znajduje się katalog pakietu `logging` (w którym z kolei znajdują się skompilowane pliki klas). Następnie należy wykonać polecenie `java PackageInfo logging`.

Aplikacja `PackageInfo` zwróci wówczas następujący wynik:

```

Nie znaleziono pakietu logging

```

Komunikat błędu jest zwracany dlatego, że metoda `getPackage()` wymaga załadowania z pakietu co najmniej jednego pliku klasy, zanim będzie mogła zwrócić obiekt klasy `Package` z opisem pakietu.

Jedynym sposobem, w jaki można wyeliminować powyższy komunikat błędu, jest załadowanie klasy z pakietu. W tym celu kod z listingu 6.7 należy uwzględnić w kodzie z listingu 6.6.

**Listing 6.7.** *Dynamiczne ładowanie klasy z pliku klasy*

```
if (args.length == 3)
try
{
    Class.forName(args[2]);
}
catch (ClassNotFoundException cnfe)
{
    System.err.println("nie można załadować " + args[2]);
    return;
}
```

Powyższy fragment kodu należy wstawić przed instrukcją `Package pkg = Package.getPackage(args[0]);`. Kod ten ładuje plik klasy o nazwie wskazanej przez trzeci argument wiersza poleceń aplikacji `PackageInfo`.

Gdy nową wersję aplikacji `PackageInfo` uruchomi się poleceniem `java PackageInfo logging 1.5 logging.File`, zwróci ona dane wynikowe przedstawione poniżej. W poleceniu wywołującym jako klasę, którą aplikacja ma załadować, wskazano klasę `File` pakietu `logging`.

```
Nazwa: logging
Tytuł implementacji: null
Dostawca implementacji: null
Wersja implementacji: null
Tytuł specyfikacji: null
Dostawca specyfikacji: null
Wersja specyfikacji: null
Upakowany: false
Exception in thread "main" java.lang.NumberFormatException: Empty version string
    at java.lang.Package.isCompatibleWith(Unknown Source)
    at PackageInfo.main(PackageInfo.java:43)
```

Tak wiele wartości `null` w danych wynikowych nie powinno dziwić, ponieważ do pakietu `logging` nie dodano żadnych informacji. Dodatkowo metoda `isCompatibleWith()` rzuca wyjątek `NumberFormatException`, ponieważ pakiet `logging` nie zawiera numeru wersji specyfikacji w zapisie z kropką (numer ten jest `null`).

Zapewne najprostszym sposobem umieszczenia informacji na temat pakietu w pakiecie `logging` jest utworzenie pliku `logging.jar` w sposób podobny do opisanego w rozdziale 4. Najpierw jednak trzeba utworzyć niewielki plik tekstowy, który będzie zawierał te informacje o pakiecie. Plik można nazwać dowolnie. Na listingu 6.8 została przedstawiona wersja pliku o nazwie `manifest.mf`.

**Listing 6.8.** *Zawartość pliku `manifest.mf` z informacjami o pakiecie*

```
Implementation-Title: Implementacja mechanizmu rejestracji
Implementation-Vendor: Jeff Friesen
Implementation-Version: 1.0a
Specification-Title: Specyfikacja mechanizmu rejestracji
Specification-Vendor: Jeff "JavaJeff" Friesen
Specification-Version: 1.0
Sealed: true
```

- 
- **Uwaga** • Należy pamiętać o tym, by ostatni wiersz pliku (`Sealed: true`) zakończyć naciśnięciem klawisza *Return* lub *Enter*. W przeciwnym razie aplikacja zwróci prawdopodobnie wartość `Upakowany: false`, ponieważ narzędzie *jar* pakietu JDK może nie zapisać w pakiecie logging odpowiadającego temu parametrowi wiersza z pliku manifestu. Potwierdza to powszechną opinię, że narzędzie *jar* jest dość kapryśne.
- 

Aby utworzyć plik JAR, który będzie zawierał pakiet logging i składające się na niego pliki oraz którego **manifest**, czyli specjalny plik *MANIFEST.MF*, będzie zawierał informacje na temat zawartości pliku JAR przedstawione na listingu 6.8, należy wykonać następujące polecenie:

```
jar cfm logging.jar manifest.mf logging
```

Powyższe polecenie utworzy plik JAR o nazwie *logging.jar* (o czym decyduje opcja `c` odpowiedzialna za utworzenie i opcja `f`, która mówi o pliku). Polecenie włączy także zawartość pliku *manifest.mf* (dzięki opcji `m` jak *manifest*) do pliku *MANIFEST.MF*, który znajduje się w katalogu *META-INF* pakietu.

- 
- **Uwaga** • Aby dowiedzieć się więcej na temat manifestu pliku JAR, warto zapoznać się z punktem „Jar Manifest” na stronie „JAR File Specification” dokumentacji JDK (<http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html#JAR%20Manifest>).
- 

Jeżeli narzędzie *jar* nie zwróci żadnych komunikatów o błędach, należy wykonać przedstawione poniżej polecenie wiersza poleceń systemu Windows (lub wiersza systemu operacyjnego, który akurat jest używany). Polecenie to wykona aplikację `PackageInfo` i odczyta informacje na temat pakietu logging:

```
java -cp logging.jar;. PackageInfo logging 1.0 logging.File
```

Tym razem wynik działania aplikacji będzie następujący:

```
Nazwa: logging
Tytuł implementacji: Implementacja mechanizmu rejestracji
Dostawca implementacji: Jeff Friesen
Wersja implementacji: 1.0a
Tytuł specyfikacji: Specyfikacja mechanizmu rejestracji
Dostawca specyfikacji: Jeff "JavaJeff" Friesen
Wersja specyfikacji: 1.0
Upakowany: false
Zgodny z wersją 1.0: true
```

## Podstawowe klasy opakowujące

Pakiet `java.lang` zawiera klasy `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` i `Short`. Są to tak zwane **podstawowe klasy opakowujące** (ang. *primitive wrapper classes*), ponieważ opakowują one wartości typów podstawowych.

- 
- **Uwaga** • Podstawowe klasy opakowujące są również znane jako **klasy wartości** (ang. *value classes*).
-

Java udostępnia osiem wspomnianych podstawowych klas opakowujących z dwóch powodów:

- Platforma kolekcji (o której więcej powiemy w rozdziale 8.) udostępnia listy, zbiory i mapy, które mogą przechowywać jedynie obiekty, natomiast nie można w nich umieszczać wartości typów podstawowych. Wartość typu podstawowego umieszcza się w instancji podstawowej klasy opakowującej, a następnie tę instancję można umieścić w kolekcji.
- Podstawowe klasy opakowujące są kontenerami, w których można wygodnie skojarzyć stałe (na przykład `MAX_VALUE` i `MIN_VALUE`) i metody klas (takie jak metody `parseInt()` klasy `Integer` i metody `isDigit()`, `isLetter()` i `toUpperCase()` klasy `Character`) z typami podstawowymi.

W tym punkcie opiszemy wszystkie podstawowe klasy opakowujące, a także klasę o nazwie `Number`.

## Klasa `Boolean`

Klasa `Boolean` to najmniejsza spośród podstawowych klas opakowujących. Klasa ta deklaruje trzy stałe, z których dwie to stałe `TRUE` i `FALSE`, oznaczające predefiniowane obiekty klasy `Boolean`.

Klasa `Boolean` deklaruje także dwa konstruktory, które inicjalizują obiekty klasy:

- `Boolean(boolean value)` inicjalizuje obiekt klasy `Boolean` wartością `value`.
- `Boolean(String s)` przekształca tekst przypisany parametrowi `s` na wartość `true` lub `false`, a następnie umieszcza tę wartość w obiekcie klasy `Boolean`.

Drugi konstruktor porównuje wartość parametru `s` z wartością `true`. Ponieważ w porównaniu tym nie bierze się pod uwagę wielkości liter, dowolna kombinacja czterech liter składających się na wartość `true` (czyli `true`, `TRUE`, `tRue` i tak dalej) zostanie umieszczona w obiekcie. W przeciwnym razie w obiekcie zostanie umieszczona wartość `false`.

Uzupełnieniem dla konstruktorów klasy `Boolean` jest metoda `boolean booleanValue()`, która zwraca opakowaną wartość klasy `Boolean`.

Klasa `Boolean` deklaruje także lub pokrywa następujące metody:

- `int compareTo(Boolean b)`, która porównuje bieżący obiekt klasy `Boolean` z parametrem `b`, aby ustalić ich kolejność. Metoda zwraca wartość 0, jeżeli bieżący obiekt zawiera tę samą wartość logiczną co parametr `b`, wartość dodatnią, jeżeli bieżący obiekt zawiera wartość `true`, a parametr `b` ma wartość `false`, oraz wartość ujemną, jeżeli bieżący obiekt zawiera wartość `false`, a parametr `b` wartość `true`.
- `boolean equals(Object o)`, która porównuje bieżący obiekt klasy `Boolean` z parametrem `o` i zwraca `true`, jeżeli `o` nie jest `null`, `o` jest typu `Boolean` i obydwa obiekty zawierają tę samą wartość logiczną.
- `static boolean getBoolean(String name)`, która zwraca `true`, jeżeli właściwość systemowa (o których powiemy w rozdziale 7.) identyfikowana przez `name` istnieje i jest równa `true`.

- `int hashCode()`, zwracająca odpowiedni kod skrótu, dzięki któremu możliwe jest wykorzystanie obiektów klasy `Boolean` w kolekcjach opartych na skrótach (powiemy o nich w rozdziale 8.).
- `static boolean parseBoolean(String s)`, która parsuje wartość parametru `s` i zwraca `true`, jeżeli `s` równa się `"true"`, `"TRUE"`, `"True"` lub dowolnej innej kombinacji tych liter. W przeciwnym razie metoda zwraca `false`. (**Parsowanie** to czynność, w trakcie której sekwencja znaków jest rozkładana na komponenty o określonym znaczeniu, tak zwane **tokeny**).
- `String toString()`, która zwraca `"true"`, jeżeli bieżąca instancja klasy `Boolean` zawiera `true`. W przeciwnym razie metoda zwraca `"false"`.
- `static String toString(boolean b)`, która zwraca `"true"`, jeżeli `b` zawiera `true`. W przeciwnym razie metoda zwraca wartość `"false"`.
- `static Boolean valueOf(boolean b)`, która zwraca `TRUE`, jeżeli `b` zawiera `true` lub `FALSE`, jeżeli `b` zawiera `false`.
- `static Boolean valueOf(String s)`, która zwraca `TRUE`, jeżeli `s` równa się `"true"`, `"TRUE"`, `"True"` lub dowolnej innej kombinacji tych liter. W przeciwnym razie metoda zwraca `FALSE`.

---

■ **Ostrzeżenie** • Programiści, którzy dopiero poznają klasę `Boolean`, często spodziewają się, że metoda `getBoolean()` zwróci wartość `true` lub `false` obiektu klasy `Boolean`. Jednak metoda `getBoolean()` zwraca wartość właściwości systemowej opartej na klasie `Boolean` (więcej na temat właściwości systemowych powiemy w rozdziale 7.). Aby uzyskać wartość `true` lub `false`, należy skorzystać z metody `booleanValue()`.

---

W większości przypadków lepiej jest użyć stałych `TRUE` lub `FALSE`, zamiast tworzyć obiekty klasy `Boolean`. Załóżmy na przykład, że trzeba zaimplementować metodę, która zwróci obiekt klasy `Boolean` z wartością `true`, jeżeli argument typu `double` tej metody będzie miał wartość ujemną, lub `false`, jeżeli argument ten będzie mieć wartość dodatnią lub będzie równy zero. Potrzebną metodę można zadeklarować w sposób analogiczny jak w przypadku metody `isNegative()` z listingu 6.9.

**Listing 6.9.** *Metoda `isNegative()`, w której tworzy się obiekt klasy `Boolean`, choć nie jest to konieczne*

```
public Boolean isNegative(double d)
{
    return new Boolean(d < 0);
}
```

Kod metody jest zwarty, lecz niepotrzebnie tworzy się w niej obiekt klasy `Boolean`. Jeśli metoda będzie często wywoływana, zostanie utworzonych wiele obiektów `Boolean`, które zajmą przestrzeń na stercie. Gdy na stercie zaczyna brakować miejsca, włączany jest proces odświeżania, który spowalnia działanie aplikacji i obniża w ten sposób jej wydajność.

Na listingu 6.10 przedstawiono ulepszoną wersję metody `isNegative()`.

**Listing 6.10.** Zrefaktoryzowana metoda `isNegative()`, w której nie tworzy się obiektów klasy `Boolean`

```
public Boolean isNegative(double d)
{
    return (d < 0) ? Boolean.TRUE : Boolean.FALSE;
}
```

W nowej wersji metody nie trzeba już tworzyć obiektów klasy `Boolean`, ponieważ metoda zwraca predefiniowany obiekt `TRUE` lub `FALSE`.

- 
- **Wskazówka** • Należy dążyć do tego, by liczba tworzonych obiektów klasy `Boolean` była jak najmniejsza. Dzięki temu mniejsza będzie ilość pamięci zużywanej przez aplikację, a wydajność samej aplikacji będzie wyższa, ponieważ proces odśmiecania nie będzie musiał być uruchamiany zbyt często.
- 

## Klasa `Character`

Klasa `Character` jest największą spośród podstawowych klas opakowujących. Zawiera znaczną liczbę stałych, konstruktor, wiele metod oraz dwie klasy zagnieżdżone (`Subset` i `UnicodeBlock`).

- 
- **Uwaga** • Wysoki stopień złożoności klasy `Character` ma swoje źródło w tym, że Java obsługuje `Unicode` (<http://pl.wikipedia.org/wiki/Unicode>). Dla uproszczenia pominięto większość mechanizmów klasy związanych z obsługą `Unicode`, ponieważ wykraczają one poza zakres tego rozdziału.
- 

Klasa `Character` deklaruje jeden konstruktor `Character(char value)`, za pomocą którego obiekt klasy `Character` inicjalizuje się wartością `value`. Uzupełnieniem konstruktora jest metoda `charValue()`, która zwraca wartość opakowanego znaku.

Programiści, którzy implementują swoje pierwsze aplikacje, często używają wyrażeń w postaci `ch >= '0' && ch <= '9'` (aby sprawdzić, czy `ch` zawiera cyfry) oraz `ch >= 'A' && ch <= 'Z'` (aby sprawdzić, czy `ch` zawiera wielką literę). Takiej praktyki powinno się jednak unikać z następujących względów:

- Bardzo łatwo jest popełnić błąd w wyrażeniu. Na przykład wyrażenie `ch > '0' && ch <= '9'` zawiera drobny błąd, który powoduje, że wartość `'0'` nie spełnia warunku wyrażenia.
- Na podstawie wyrażeń stosunkowo trudno jest ustalić, co tak naprawdę jest w nich sprawdzane.
- W wyrażeniach można używać jedynie cyfr łacińskich (0 – 9) oraz łacińskich liter (A – Z i a – z). W wyrażeniach nie są natomiast uwzględniane cyfry i litery, które są poprawne w innych językach. Na przykład `'\u0bbeb'` to literał znakowy, który reprezentuje jedną z cyfr w języku tamilijskim.

Klasa `Character` deklaruje kilka metod narzędziowych, które służą do porównywania oraz przekształcania i w ten sposób rozwiązują przytoczone przed chwilą problemy. Metodami tymi są:



- `static boolean isDigit(char ch)`, która zwraca `true`, jeśli `ch` zawiera cyfrę (standardowo z przedziału od 0 do 9, ale również cyfry z innych języków).
- `static boolean isLetter(char ch)`, która zwraca `true`, jeśli `ch` zawiera literę (standardowo `A – Z` lub `a – z`, ale również litery z innych języków).
- `static boolean isLetterOrDigit(char ch)`, która zwraca `true`, jeśli `ch` zawiera literę lub cyfrę (standardowo `A – Z`, `a – z` lub `0 – 9`, ale również litery lub cyfry z innych języków).
- `static boolean isLowerCase(char ch)`, która zwraca `true`, jeśli `ch` zawiera małą literę.
- `static boolean isUpperCase(char ch)`, która zwraca `true`, jeśli `ch` zawiera wielką literę.
- `static boolean isWhitespace(char ch)`, która zwraca `true`, jeśli `ch` zawiera znak niewidoczny (zazwyczaj znak spacji, tabulacji poziomej, powrotu karetki lub nowego wiersza).
- `static char toLowerCase(char ch)`, która zwraca małą literę odpowiadającą wielkiej literze zawartej w `ch`. Jeżeli `ch` nie zawiera wielkiej litery, metoda zwraca wartość `ch`.
- `static char toUpperCase(char ch)`, która zwraca wielką literę odpowiadającą małej literze zawartej w `ch`. Jeżeli `ch` nie zawiera małej litery, metoda zwraca wartość `ch`.

Na przykład zamiast `ch >= '0' && ch <= '9'` zdecydowanie lepiej jest wykonać metodę `isDigit(ch)`, ponieważ w ten sposób unika się ryzyka popełnienia błędów, sama metoda jest zdecydowanie bardziej czytelna, a poza tym zwraca ona wartość `true` nie tylko dla cyfr łańcuchowych, ale także cyfr z innych języków (na przykład `'\u094d'`).

## Klasy `Float` i `Double`

Klasy `Float` i `Double` przechowują w obiektach typu `Float` i `Double` odpowiednio wartości zmiennopozycyjne i wartości zmiennopozycyjne o podwójnej precyzji. Klasy te deklarują następujące stałe:

- `MAX_VALUE` identyfikuje maksymalną wartość, którą można reprezentować jako wartość typu `float` lub `double`.
- `MIN_VALUE` identyfikuje minimalną wartość, którą można reprezentować jako wartość typu `float` lub `double`.
- `NaN` reprezentuje `0.0F/0.0F` jako typu `float` oraz `0.0/0.0` jako typu `double`.
- `NEGATIVE_INFINITY` reprezentuje minus nieskończoność jako typu `float` lub `double`.
- `POSITIVE_INFINITY` reprezentuje plus nieskończoność jako typu `float` lub `double`.

Klasy `Float` i `Double` deklarują także następujące konstruktory, które inicjalizują obiekty tych klas:

- `Float(float value)` inicjalizuje obiekt klasy `Float` wartością `value`.
- `Float(double value)` inicjalizuje obiekt klasy `Float` odpowiednikiem wartości `value` typu `float`.

- `Float(String s)` przekształca tekst ze zmiennej `s` w wartość zmiennopozycyjną i inicjalizuje tą wartością obiekt klasy `Float`.
- `Double(double value)` inicjalizuje obiekt klasy `Double` wartością `value`.
- `Double(String s)` przekształca tekst ze zmiennej `s` w wartość zmiennopozycyjną o podwójnej precyzji i inicjalizuje tą wartością obiekt klasy `Double`.

Uzupełnieniem dla konstruktorów klasy `Float` jest metoda `float floatValue()`, która zwraca opakowaną wartość zmiennopozycyjną. Analogicznie uzupełnieniem dla konstruktorów klasy `Double` jest metoda `double doubleValue()`, która zwraca opakowaną wartość zmiennopozycyjną o podwójnej precyzji.

Oprócz metody `floatValue()` klasa `Float` deklaruje kilka metod narzędziowych. Są to następujące metody:

- `static int floatToIntBits(float value)` przekształca `value` w 32-bitową liczbę całkowitą.
- `static boolean isInfinite(float f)` zwraca `true`, jeśli wartością `f` jest plus nieskończoność lub minus nieskończoność. Podobna do niej metoda `public boolean isInfinite()` zwraca `true`, jeśli wartością bieżącego obiektu klasy `Float` jest plus nieskończoność lub minus nieskończoność.
- `static boolean isNaN(float f)` zwraca `true`, jeśli wartością `f` jest `NaN`. Podobna do niej metoda `public boolean isNaN()` zwraca `true`, jeśli wartością bieżącego obiektu klasy `Float` jest `NaN`.
- `static float parseFloat(String s)` parsuje `s` i zwraca wartość zmiennopozycyjną, która odpowiada zawartej w `s` tekstowej reprezentacji wartości zmiennopozycyjnej lub rzuca wyjątek `NumberFormatException`, jeżeli reprezentacja ta jest nieprawidłowa (na przykład zawiera litery).

Oprócz metody `doubleValue()` klasa `Double` deklaruje kilka metod narzędziowych. Są to następujące metody:

- `static long doubleToLongBits(double value)` przekształca `value` na długą liczbę całkowitą.
- `static boolean isInfinite(double d)` zwraca `true`, jeśli wartością `d` jest plus nieskończoność lub minus nieskończoność. Podobna do niej metoda `boolean isInfinite()` zwraca `true`, jeśli wartością bieżącego obiektu klasy `Double` jest plus nieskończoność lub minus nieskończoność.
- `static boolean isNaN(double d)` zwraca `true`, jeśli wartością `d` jest `NaN`. Podobna do niej metoda `public boolean isNaN()` zwraca `true`, jeśli wartością bieżącego obiektu klasy `Double` jest `NaN`.
- `static double parseDouble(String s)` parsuje `s` i zwraca wartość zmiennopozycyjną o podwójnej precyzji, która odpowiada zawartej w `s` tekstowej reprezentacji wartości zmiennopozycyjnej o podwójnej precyzji lub rzuca wyjątek `NumberFormatException`, jeżeli reprezentacja ta jest nieprawidłowa.

Za pomocą metod `floatToIntBits()` i `doubleToIntBits()` implementuje się metody `equals()` i `hashCode()`, które mają dotyczyć pól typu `float` i `double`. Dzięki metodom `floatToIntBits()` i `doubleToIntBits()` metody `equals()` i `hashCode()` działają prawidłowo w opisanych poniżej sytuacjach.

- `equals()` musi zwrócić `true`, jeśli `f1` i `f2` zawierają `Float.NaN` (lub jeśli `d1` i `d2` zawierają `Double.NaN`). Gdyby metoda `equals()` była zaimplementowana w sposób analogiczny do `f1.floatValue() == f2.floatValue()` (albo `d1.doubleValue() == d2.doubleValue()`), metoda zwróciłaby `false`, ponieważ `NaN` nie jest równa żadnej innej wartości oprócz samej siebie.
- `equals()` musi zwrócić `false`, jeśli `f1` zawiera `+0.0` i `f2` zawiera `-0.0` (albo odwrotnie) lub jeśli `d1` zawiera `+0.0` i `d2` zawiera `-0.0` (albo odwrotnie). Gdyby metoda `equals()` była zaimplementowana w sposób analogiczny do `f1.floatValue() == f2.floatValue()` (lub `d1.doubleValue() == d2.doubleValue()`), metoda zwróciłaby `true`, ponieważ wyrażenie `+0.0 == -0.0` zwraca `true`.

Spełnienie tych wymagań jest niezbędne, aby kolekcje bazujące na skrótach (o których więcej powiemy w rozdziale 8.) funkcjonowały prawidłowo. Na listingu 6.11 widać, w jaki sposób opisane wymagania wpływają na działanie metod `equals()` klas `Float` i `Double`.

**Listing 6.11.** Ilustracja działania metody `equals()` klasy `Float` w kontekście wartości `NaN` oraz metody `equals()` klasy `Double` w kontekście wartości `+0.0` i `-0.0`

```
public static void main(String[] args)
{
    Float f1 = new Float(Float.NaN);
    System.out.println(f1.floatValue());
    Float f2 = new Float(Float.NaN);
    System.out.println(f2.floatValue());
    System.out.println(f1.equals(f2));
    System.out.println(Float.NaN == Float.NaN);
    System.out.println();
    Double d1 = new Double(+0.0);
    System.out.println(d1.doubleValue());
    Double d2 = new Double(-0.0);
    System.out.println(d2.doubleValue());
    System.out.println(d1.equals(d2));
    System.out.println(+0.0 == -0.0);
}
```

Gdy aplikacja zostanie uruchomiona, zwróci przedstawione poniżej dane wynikowe. Dowodzą one, że metoda `equals()` klasy `Float` odpowiednio obsługuje wartość `NaN`, a metoda `equals()` klasy `Double` odpowiednio uwzględnia wartości `+0.0` i `-0.0`:

```
NaN
NaN
true
false
```

```
0.0
-0.0
false
true
```

- **Wskazówka** • Aby sprawdzić, czy wartości typu float lub double są równe plus nieskończoności lub minus nieskończoności (ale nie obydwoóm tym wartościom), nie należy do tego celu używać metody `isInfinite()`. Zamiast niej wartość należy porównać za pomocą operatora `==` ze stałą `NEGATIVE_INFINITY` lub `POSITIVE_INFINITY`, na przykład `f == Float.NEGATIVE_INFINITY`.

Metody `parseFloat()` i `parseDouble()` okażą się przydatne w wielu sytuacjach. Na przykład na listingu 6.12 wykorzystano metodę `parseDouble()`, aby wykonać parsowanie argumentów wiersza poleceń do wartości typu `double`.

**Listing 6.12.** Parsowanie argumentów wiersza poleceń do wartości zmiennopozycyjnych o podwójnej precyzji

```
public static void main(String[] args)
{
    if (args.length != 3)
    {
        System.err.println("użycie: java Calc wartość1 operator wartość2");
        System.err.println("operatorem może być +, -, * lub /");
        return;
    }
    try
    {
        double value1 = Double.parseDouble(args[0]);
        double value2 = Double.parseDouble(args[2]);
        if (args[1].equals("+"))
            System.out.println(value1+value2);
        else
            if (args[1].equals("-"))
                System.out.println(value1-value2);
            else
                if (args[1].equals("*"))
                    System.out.println(value1*value2);
                else
                    if (args[1].equals("/"))
                        System.out.println(value1/value2);
                    else
                        System.err.println("nieprawidłowy operator: " + args[1]);
    }
    catch (NumberFormatException nfe)
    {
        System.err.println("Nieprawidłowy format liczby: " + nfe.getMessage());
    }
}
```

Aby wypróbować powyższą aplikację, można wywołać ją poleceniem `java Calc 10E+3 + 66.0`. Wynikiem zwróconym przez aplikację będzie `10066.0`. Gdyby natomiast wywołać ją poleceniem `java Calc 10E+3 + A`, aplikacja zwróciłaby następujący komunikat błędu: Nieprawidłowy format liczby: "A". Komunikat ten jest spowodowany rzuceniem wyjątku `NumberFormatException`  $\rightarrow$  `Exception`, co zachodzi w wyniku drugiego wywołania metody `parseDouble()`.

Choć klasa `NumberFormatException` opisuje wyjątki niekontrolowane, a wyjątków niekontrolowanych zwykle się nie obsługuje, ponieważ reprezentują one błędy w kodzie źródłowym, to jednak w tym przykładzie `NumberFormatException` nie pasuje do tego schematu. Wyjątek jest rzucany nie w wyniku błędu w kodzie źródłowym, lecz z powodu przekazania do aplikacji nieprawidłowego argumentu liczbowego, na co jakość kodu źródłowego nie ma żadnego wpływu.

## Klasy Integer, Long, Short i Byte

Klasy Integer, Long, Short i Byte przechowują wartości całkowitoliczbowe odpowiednio 32-, 64-, 16- i 8-bitowe w obiektach typów odpowiednio Integer, Long, Short i Byte.

Każda z tych klas deklaruje stałe `MAX_VALUE` i `MIN_VALUE` identyfikujące wartości maksymalną i minimalną, które może reprezentować typ podstawowy powiązany z daną klasą.

Klasy deklarują także konstruktory, które inicjalizują ich obiekty.

- `Integer(int value)` inicjalizuje obiekt klasy Integer wartością `value`.
- `Integer(String s)` przekształca tekst w argumencie `s` na 32-bitową wartość całkowitoliczbową i inicjalizuje tą wartością obiekt klasy Integer.
- `Long(long value)` inicjalizuje obiekt klasy Long wartością `value`.
- `Long(String s)` przekształca tekst w argumencie `s` na 64-bitową wartość całkowitoliczbową i inicjalizuje tą wartością obiekt klasy Long.
- `Short(short value)` inicjalizuje obiekt klasy Short wartością `value`.
- `Short(String s)` przekształca tekst w argumencie `s` na 16-bitową wartość całkowitoliczbową i inicjalizuje tą wartością obiekt klasy Short.
- `Byte(byte value)` inicjalizuje obiekt klasy Byte wartością `value`.
- `Byte(String s)` przekształca tekst w argumencie `s` na 8-bitową wartość całkowitoliczbową i inicjalizuje tą wartością obiekt klasy Byte.

Uzupełnieniem dla konstruktorów klasy Integer jest metoda `int intValue()`, dla konstruktorów klasy Long takim uzupełnieniem jest metoda `long longValue()`, dla konstruktorów klasy Short — metoda `short shortValue()`, zaś dla konstruktorów klasy Byte — metoda `byte byteValue()`. Wszystkie wspomniane metody zwracają opakowane liczby całkowite.

Wszystkie cztery klasy deklarują różnorodne przydatne metody do przetwarzania liczb całkowitych. Na przykład klasa Integer deklaruje wymienione poniżej metody klas, które przeprowadzają konwersję 32-bitowej liczby całkowitej na ciąg znaków String zgodnie ze wskazaną reprezentacją (binarną, szesnastkową, ósemkową i dziesiętkową):

- `static String toBinaryString(int i)` zwraca obiekt klasy String, który zawiera binarną reprezentację argumentu `i`. Na przykład `Integer.toBinaryString(255)` zwróci obiekt klasy String, który będzie zawierać ciąg `11111111`.
- `static String toHexString(int i)` zwraca obiekt klasy String, który zawiera szesnastkową reprezentację argumentu `i`. Na przykład `Integer.toHexString(255)` zwróci obiekt klasy String, który będzie zawierać ciąg `ff`.
- `static String toOctalString(int i)` zwraca obiekt klasy String, który zawiera ósemkową reprezentację argumentu `i`. Na przykład `toOctalString(64)` zwróci obiekt klasy String, który będzie zawierać ciąg `377`.
- `static String toString(int i)` zwraca obiekt klasy String, który zawiera dziesiętkową reprezentację argumentu `i`. Na przykład `toString(255)` zwróci obiekt klasy String, który będzie zawierać ciąg `255`.

Często wygodnie jest poprzedzać ciąg binarny zerami, tak aby ciągi te móc umieszczać w kolumnach o takiej samej szerokości. Na przykład można zaimplementować aplikację, która będzie wyświetlać następujący, obustronnie wyrównany zbiór danych wynikowych:

```
11110001
+
00000111
-----
11111000
```

Niestety, metoda `toBinaryString()` nie ułatwia wykonania tego zadania. Na przykład metoda `Integer.toBinaryString(7)` zwróci obiekt klasy `String`, który będzie zawierał ciąg 111, a nie 00000111. Na listingu 6.13 przedstawiono metodę `toAlignedBinaryString()`, która wyświetla ciąg binarny w pożądanym, obustronnie wyrównanym formacie.

**Listing 6.13.** Wyrównywanie binarnego ciągu znaków

```
public static void main(String[] args)
{
    System.out.println(toAlignedBinaryString(7, 8));
    System.out.println(toAlignedBinaryString(255, 16));
    System.out.println(toAlignedBinaryString(255, 7));
}
static String toAlignedBinaryString(int i, int numBits)
{
    String result = Integer.toBinaryString(i);
    if (result.length() > numBits)
        return null; // nie można zmieścić wyniku w numBits kolumnach
    int numLeadingZeros = numBits - result.length();
    String zerosPrefix = "";
    for (int j = 0; j < numLeadingZeros; j++)
        zerosPrefix += "0";
    return zerosPrefix + result;
}
```

Metoda `toAlignedBinaryString()` przyjmuje dwa argumenty. Pierwszy zawiera 32-bitową liczbę całkowitą, którą należy przekształcić na binarny ciąg znaków, zaś drugi argument wskazuje liczbę kolumn bitów, w których należy wyświetlić ten ciąg znaków.

Najpierw jest wywoływana metoda `toBinaryString()`, która zwraca binarny ciąg znaków stanowiący odpowiednik wartości `i`, lecz bez poprzedzających zer. Następująca po nim metoda `toAlignedBinaryString()` sprawdza, czy wszystkie cyfry binarnego ciągu znaków zmieszczą się w kolumnach bitów, których liczbę wyznacza `numBits`. Jeżeli cyfr będzie więcej niż kolumn, metoda zwróci `null`. (Więcej informacji na temat metody `length()` oraz innych metod klasy `String` przedstawimy w rozdziale 7.).

W dalszej kolejności metoda `toAlignedBinaryString()` oblicza, iloma zerami należy poprzedzić wartość `result`, po czym w pętli tworzy ciąg znaków złożony z takiej liczby zer. Na koniec metoda zwraca ciąg zer, które mają poprzedzić wynikowy ciąg znaków.

Złożona operacja połączenia ciągów znaków za pomocą operatora przypisania (`+=`) na pierwszy rzut oka nie budzi wątpliwości, jednak tak naprawdę jest to rozwiązanie mało wydajne, ponieważ pośrednie obiekty tworzone w trakcie tej operacji są najpierw tworzone, a następnie niszczone. Zastosowaliśmy jednak tak niewydajny kod, aby w rozdziale 7. przeciwstawić mu rozwiązanie zdecydowanie lepsze pod względem wydajności.

Gdy aplikacja zostanie uruchomiona, zwróci następujące dane wynikowe:

```
00000111
0000000011111111
null
```

## Klasa Number

Wszystkie spośród klas `Float`, `Double`, `Integer`, `Long`, `Short` i `Byte` udostępniają nie tylko metodę `xValue()` odnoszącą się do samej klasy, ale również metody `xValue()` odnoszące się do wszystkich pozostałych klas. Na przykład w klasie `Float` dostępne są, oprócz `floatValue()`, również metody `doubleValue()`, `intValue()`, `longValue()`, `shortValue()` oraz `byteValue()`.

Wszystkie sześć metod to składowe klasy `java.lang.Number`, która jest abstrakcyjną klasą bazową dla klas `Float`, `Double`, `Integer`, `Long`, `Short` i `Byte`. Metody `floatValue()`, `doubleValue()`, `intValue()` i `longValue()` klasy `Number` są więc metodami abstrakcyjnymi. Klasa `Number` jest również klasą bazową dla klas `BigDecimal` i `BigInteger` (a także dla niektórych klas przeznaczonych do pracy współbieżnej, o czym więcej powiemy w rozdziale 9.).

Klasa `Number` istnieje po to, by uprościć iterowanie przez kolekcję obiektów klas potomnych po `Number`. Na przykład można zadeklarować zmienną typu `List<Number>` i zainicjalizować ją instancją klasy `ArrayList<Number>`. W tak utworzonej kolekcji można przechowywać różne klasy potomne po `Number` i iterować przez kolekcję, wykorzystując do tego wielopostaciowość wywołań metod klas podrzędnych.

## API References

W rozdziale 2. opisano mechanizm odśmiecania, który usuwa obiekt ze sterty, jeśli nie istnieją już żadne odwołania do tego obiektu.

W rozdziale 3. z kolei przedstawiono metodę `finalize()` klasy `Object`. Jest to metoda, którą wywołuje proces odśmiecania, zanim usunie obiekt ze sterty. Metoda `finalize()` pozwala obiektowi na wykonanie czynności sprzątających.

W tym punkcie będzie kontynuowany wątek zapoczątkowany w rozdziałach 2. oraz 3. Zostanie też zaprezentowane API References. Interfejs ten umożliwia aplikacji komunikowanie się, w ograniczonym zakresie, z procesem odśmiecania.

Na początek przedstawimy podstawowe pojęcia związane z API References. Następnie opiszemy klasy `Reference` i `ReferenceQueue` interfejsu, po czym zakończymy prezentacją klas `SoftReference`, `WeakReference` i `PhantomReference`.

## Podstawowe pojęcia

Gdy uruchamia się aplikację, tworzony jest dla niej **główny zbiór odwołań** (ang. *root set of references*). Zbiór ten jest kolekcją zmiennych lokalnych, parametrów, pól klas i pól instancji, które aktualnie istnieją i zawierają — potencjalnie puste — odwołania do obiektów. Zbiór główny zmienia się z upływem czasu i w miarę wykonywania przez aplikację kolejnych zadań. Na przykład po powrocie z metody ze zbioru znikają jej parametry.

Wiele procesów odśmiecania odczytuje główny zbiór odwołań w momencie, gdy zostają uruchomione. Na podstawie głównego zbioru odwołań proces odśmiecania sprawdza, czy obiekt jest **osiągalny** (albo czy obiekt **żyje**, to znaczy czy istnieją jakieś odwołania do niego), czy też jest **nieosiągalny** (czyli nie istnieją żadne odwołania do niego). Proces odśmiecania nie może niszczyć obiektów, które są osiągalne — proces ten może usuwać ze sterty jedynie te obiekty, które, począwszy od głównego zbioru odwołań, nie są osiągalne.

---

■ **Uwaga** • Obiektami osiągalnymi są także te obiekty, które są osiągalne pośrednio za pośrednictwem zmiennych ze zbioru głównego. Inaczej mówiąc, chodzi o obiekty, które są osiągalne za pośrednictwem żywych obiektów osiągalnych bezpośrednio za pomocą tych zmiennych. Obiekt, który pozostaje nieosiągalny na żadnej ścieżce, bez względu na to, którą zmienną z głównego zbioru odwołań się wybierze, podlega przetworzeniu przez mechanizm odśmiecania.

---

Począwszy od wersji 1.2 języka Java, obiekty osiągalne dzieli się na obiekty silnie osiągalne, miękko osiągalne, słabo osiągalne i złudnie osiągalne. W odróżnieniu od obiektów silnie osiągalnych, obiekty osiągalne miękko, słabo i złudnie mogą podlegać przetwarzaniu przez mechanizm odśmiecania.

W poniższych punktach scharakteryzowano cztery wspomniane typy osiągalności zależnie od mocy odwołania — od najsilniejszego do najsłabszego.

- Obiekt jest **silnie osiągalny** (ang. *strongly reachable*), jeżeli jest osiągalny dla wątku i wątek ten nie musi przechodzić przez obiekty API Reference — wątek korzysta z **silnego odwołania** (ang. *strong reference*) przypisanego zmiennej z głównego zbioru odwołań. Nowo utworzony obiekt (na przykład obiekt, do którego odwołuje się zmienna `d` z instrukcji `Double d = new Double(1.0);`) jest silnie osiągalny dla wątku, który ten obiekt stworzył. (Więcej na temat wątków powiemy w rozdziale 7.).
- Obiekt jest **miękko osiągalny** (ang. *softly reachable*), jeżeli nie jest silnie osiągalny, lecz staje się osiągalny za pośrednictwem **odwołania miękkiego** (ang. *soft reference*), czyli odwołania do obiektu, które jest przechowywane w obiekcie klasy `SoftReference`. Najsilniejszym odwołaniem do takiego obiektu jest odwołanie miękkie. Gdy ilość miejsca na stercie jest już niewielka, proces odśmiecania zwykle usuwa miękkie odwołania do najstarszych miękko osiągalnych obiektów, po czym finalizuje te obiekty (przez wywołanie metody `finalize()`) i je usuwa.
- Obiekt jest **słabo osiągalny** (ang. *weakly reachable*), jeżeli nie jest silnie ani miękko osiągalny, natomiast staje się osiągalny za pośrednictwem **odwołania słabego** (ang. *weak reference*), czyli odwołania do obiektu, które jest przechowywane w obiekcie klasy `WeakReference`. Najsilniejszym odwołaniem do takiego obiektu jest odwołanie słabe. Proces odśmiecania usuwa słabe odwołania do słabo osiągalnych obiektów i niszczy je (a wcześniej finalizuje) przy następnym uruchomieniu procesu, nawet jeśli w pamięci jest dużo wolnego miejsca.
- Obiekt jest **złudnie osiągalny** (ang. *phantom reachable*), jeżeli nie jest silnie, miękko ani słabo osiągalny, został sfinalizowany i proces odśmiecania jest gotowy na odzyskanie zajmowanej przez niego pamięci. Dodatkowo do obiektu złudnie osiągalnego



wiedzie **złudne odwołanie** (ang. *phantom reference*), czyli odwołanie do obiektu przechowywane w obiekcie klasy `PhantomReference`. Najsilniejszym odwołaniem do obiektu złudnie osiągalnego jest odwołanie **złudne**.

- 
- **Uwaga** • Jedyna różnica między odwołaniem miękkim a słabym sprowadza się do tego, że prawdopodobieństwo usunięcia obiektu miękko osiągalnego przez proces odświeczania jest niższe niż w przypadku obiektu słabo osiągalnego. Poza tym odwołanie słabe nie ma wystarczającej mocy, by utrzymać obiekt w pamięci.
- 

Obiekty, do których odwołanie jest przechowywane w obiektach klas `SoftReference`, `WeakReference` lub `PhantomReference`, to tak zwane **referenty**.

## Klasy `Reference` i `ReferenceQueue`

API `Reference` zawiera pięć klas, które wchodzą w skład pakietu `java.lang.ref`. Jądrzem tego pakietu są klasy `Reference` i `ReferenceQueue`.

Klasa `Reference` jest abstrakcyjną klasą główną dla wchodzących w skład pakietu klas potomnych `SoftReference`, `WeakReference` i `PhantomReference`.

Klasa `ReferenceQueue` to klasa, której instancje opisują strukturę kolejek danych. Jeśli instancja klasy `ReferenceQueue` zostanie skojarzona z obiektem klasy potomnej `Reference` (czyli, krócej mówiąc, z obiektem `Reference`), wówczas w momencie gdy referent, do którego wiedzie odwołanie enkapsulowane w obiekcie `Reference`, zostanie zniszczony przez proces odświeczania, obiekt `Reference` zostanie dodany do kolejki.

- 
- **Uwaga** • Obiekt klasy `ReferenceQueue` kojarzy się z obiektem `Reference` w ten sposób, że obiekt klasy `ReferenceQueue` przekazuje się do konstruktora odpowiedniej klasy potomnej po klasie `Reference`.
- 

Klasa `Reference` jest zadeklarowana jako ogólny typ `Reference<T>`, w którym `T` oznacza typ referenta. Klasa udostępnia następujące metody:

- `void clear()` przypisuje odwołaniu `null`. Obiekt klasy `Reference`, na którym jest wywoływana ta metoda, nie jest kolejgowany (wstawiany) do skojarzonej z nim kolejki odwołań (jeżeli oczywiście taka kolejka odwołań jest skojarzona z obiektem). (Proces odświeczania części odwołania w sposób bezpośredni, a nie przez wywołanie metody `clear()`. Metoda `clear()` jest wywoływana przez aplikację).
- `boolean enqueue()` dodaje obiekt klasy `Reference`, na którym metoda została wywołana, do skojarzonej z nią kolejki odwołań. Metoda zwraca `true`, gdy obiekt klasy `Reference` został zakolejkowany, lub `false` w przeciwnym wypadku — wówczas jest to znak, że obiekt został zakolejkowany już wcześniej albo w momencie jego tworzenia nie został skojarzony z żadną kolejką. (Proces odświeczania kolejkuje obiekty klasy `Reference` w sposób bezpośredni, a nie przez wywołanie metody `enqueue()`. Metoda `enqueue()` jest wywoływana przez aplikację).

- `T get()` zwraca odwołanie przechowywane w obiekcie klasy `Reference`. Jeżeli przechowywane odwołanie zostało wcześniej wyczyszczone czy to przez aplikację, czy to przez proces odśmiecania, zwracaną wartością jest `null`.
- `boolean isEnqueued()` zwraca `true`, jeśli obiekt klasy `Reference` został zakolejkowany przez aplikację lub przez proces odśmiecania. W przeciwnym razie metoda zwraca wartość `false`, co oznacza, że obiekt klasy `Reference` nie został w momencie jego utworzenia skojarzony z żadną kolejką.

---

■ **Uwaga** • Klasa `Reference` deklaruje także konstruktory. Ponieważ konstruktory te są prywatne w ramach pakietu, klasami potomnymi po `Reference` mogą być wyłącznie klasy z pakietu `java.lang.ref`. Nałożenie takiego ograniczenia było konieczne dlatego, że instancje klas potomnych po `Reference` muszą ściśle współpracować z procesem odśmiecania.

---

Klasa `ReferenceQueue` jest zadeklarowana jako ogólny typ `ReferenceQueue<T>`, w którym `T` wskazuje typ referenta. Klasa `ReferenceQueue` deklaruje konstruktor i metody opisane poniżej.

- `ReferenceQueue()` inicjalizuje nową instrukcję klasy `ReferenceQueue`.
- `Reference<? extends T> poll()` sprawdza, czy w kolejce jest dostępny obiekt klasy `Reference`. Jeżeli taki obiekt jest dostępny, zostaje on usunięty z kolejki i zwrócony przez metodę. W przeciwnym razie następuje natychmiastowy powrót z metody z wartością wynikową `null`.
- `Reference<? extends T> remove()` usuwa z kolejki następny obiekt klasy `Reference` i go zwraca. Przez nieokreślony czas metoda czeka na to, by obiekt klasy `Reference` stał się dostępny, a gdy to oczekiwanie zostanie przerwane, rzuca wyjątek klasy `java.lang.InterruptedException`.
- `Reference<? extends T> remove(long timeout)` usuwa z kolejki następny obiekt klasy `Reference` i go zwraca. Metoda czeka, aż obiekt klasy `Reference` stanie się dostępny lub upłynie tyle milisekund, ile wskazuje parametr `timeout`. Wywołanie metody z argumentem `0` spowoduje, że metoda będzie oczekiwać bez końca. Jeżeli upłynie czas wskazywany przez `timeout`, metoda zwróci `null`. Jeżeli wartość `timeout` będzie ujemna, metoda rzuci wyjątek `java.lang.IllegalArgumentException`, a jeżeli jej oczekiwanie zostanie przerwane, zostanie rzucony wyjątek klasy `InterruptedException`.

## Klasa `SoftReference`

Klasa `SoftReference` opisuje obiekt klasy `Reference`, którego referent jest miękko osiągalny. Jest to klasa ogólna, która oprócz tego, że dziedziczy metody klasy `Reference` i pokrywa jej metodę `get()`, udostępnia także opisane poniżej konstruktory przeznaczone do inicjalizowania obiektów klasy `SoftReference`.

- `SoftReference(T r)` umieszcza w obiekcie odwołanie przekazane jako argument `r`. Obiekt klasy `SoftReference` zachowuje się jak miękkie odwołanie do `r`. Żaden obiekt klasy `ReferenceQueue` nie jest skojarzony z obiektem klasy `SoftReference`.

- `SoftReference(T r, ReferenceQueue<? super T> q)` umieszcza w obiekcie odwołanie przekazane jako argument `r`. Obiekt klasy `SoftReference` zachowuje się jak miękkie odwołanie do `r`. Z obiektem klasy `SoftReference` zostaje skojarzona kolejka `SoftReference` wskazywana przez `q`. Jeżeli przekazany do metody argument `q` będzie `null`, będzie to oznaczać, że z miękkim odwołaniem nie jest skojarzona żadna kolejka.

Klasa `SoftReference` przydaje się do implementowania buforów pamięci podręcznej, na przykład do przechowywania obrazków. Bufor obrazków przechowuje w pamięci obrazki (ponieważ ich załadowanie z dysku zabiera czas), a także zapewnia, że do pamięci nie trafiają duplikaty obrazków, które potencjalnie mogą mieć duże rozmiary.

Bufor obrazków zawiera odwołania do obiektów obrazków, które znajdują się już w pamięci. Gdyby były to silne odwołania, obrazki pozostawałyby w pamięci. Trzeba by wówczas ustalać, które z tych obrazków nie są już więcej potrzebne, i usuwać je z pamięci, aby mogły zostać przetworzone przez proces odświeżania.

Gdyby konieczne było ręczne usuwanie obrazków, oznaczałoby to tak naprawdę powielanie zadań wykonywanych przez proces odświeżania. Jeżeli jednak odwołania do obiektów obrazków opakuje się w obiektach klasy `SoftReference`, proces odświeżania sam ustali, kiedy obiekty te należy usunąć (zazwyczaj będzie to wówczas, gdy na sterckie będzie już mało wolnego miejsca), a także sam je usunie.

Na listingu 6.14 pokazano, w jaki sposób za pomocą klasy `SoftReference` można utrzymać podręczny bufor obrazków.

**Listing 6.14.** *Utrzymywanie podręcznego bufora obrazków*

```
class Image
{
    private byte[] image;
    private Image(String name)
    {
        image = new byte[1024*100];
    }
    static Image getImage(String name)
    {
        return new Image(name);
    }
}
public class ImageCache
{
    final static int NUM_IMAGES = 200;
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
        String[] imageNames = new String[NUM_IMAGES];
        for (int i = 0; i < imageNames.length; i++)
            imageNames[i] = new String("obrazek" + i + ".gif");

        SoftReference<Image>[] cache = new SoftReference[imageNames.length];
        for (int i = 0; i < cache.length; i++)
            cache[i] = new SoftReference<Image>(Image.getImage(imageNames[i]));

        for (int i = 0; i < cache.length; i++)
        {
            Image im = cache[i].get();
        }
    }
}
```

```

        if (im == null)
        {
            System.out.println(imageNames[i] + " nie znajduje się w buforze");
            im = Image.getImage(imageNames[i]);
            cache[i] = new SoftReference<Image>(im);
        }
        System.out.println("Rysowanie obrazka");
        im = null; // Usunięcie silnego odwołania do obrazka.
    }
}
}

```

Na listingu zadeklarowano klasę Image, która symuluje ładowanie obrazka. Każda instancja obrazka jest tworzona przez wywołanie metody klasy getImage(). Prywatna tablica image instancji zajmuje 100 KB pamięci.

Metoda main() najpierw tworzy tablicę obiektów klasy String, które zawierają nazwy plików obrazków. Sposób tworzenia tej tablicy jest daleki od wydajnego, a rozwiązanie alternatywne o odpowiedniej wydajności zostanie przedstawione w rozdziale 7.

Metoda main() tworzy następnie tablicę obiektów klasy SoftReference, która odgrywa rolę podręcznego bufora dla obiektów klasy Image. Tablica jest inicjalizowana obiektami klasy SoftReference, z których każdy jest inicjalizowany odwołaniem do obiektu klasy Image.

Na tym etapie metoda main() rozpoczyna wykonanie głównej pętli aplikacji. Pętla iteruje przez bufor i odczytuje kolejne obiekty klasy Image lub null, jeśli proces odświeżania wyczerpał miękkie odwołanie do obiektu obrazka (aby zwolnić miejsce zajmowane przez niego na sterckie).

Jeżeli odwołaniem przypisanym im nie jest null, oznacza to, że obiekt nie stał się nieosiągalny — następujący zaraz potem kod może więc wyrysować obrazek na ekranie. Instrukcja przypisania im = null; usuwa silne odwołanie do obiektu klasy Image ze zmiennej im głównego zbioru odwołań.

- 
- **Uwaga** • Wykonanie instrukcji przypisania im = null; nie jest w tej aplikacji konieczne, ponieważ zaraz w następnej iteracji im jest nadpisywane przez wartość zwróconą przez metodę get() albo pętla i aplikacja zostają zakończone. W przykładowym kodzie instrukcja ta została umieszczona po to, aby pokazać, w jaki sposób można się pozbyć wartości zmiennej im. Działanie takie okaże się przydatne, gdy wartość zmiennej pozostanie w pamięci na dłużej, jeśli przykładowa aplikacja zostanie rozbudowana do bardziej złożonej postaci, a proces odświeżania nie będzie mógł usunąć ze sterty obiektu klasy Image wskazywanego przez im ze względu na fakt, że obiekt ten będzie silnie osiągalny.
- 

Jeśli odwołanie przypisane zmiennej im będzie null, będzie to oznaczać, że obiekt klasy Image stał się nieosiągalny i prawdopodobnie został usunięty ze sterty. W takiej sytuacji obiekt obrazka trzeba ponownie utworzyć i umieścić w nowym obiekcie klasy SoftReference, który powinien następnie trafić do bufora podręcznego.

Poniżej znajduje się krótki wycinek danych wynikowych zwróconych przez aplikację — jej kod być może trzeba będzie odpowiednio dostosować, aby uzyskać takie same wyniki:

```

obrazek162.gif nie znajduje się w buforze
Rysowanie obrazka
obrazek163.gif nie znajduje się w buforze
Rysowanie obrazka
Rysowanie obrazka

```

Komunikat `Rysowanie obrazka` w ostatnim wierszu przytoczonych danych wynikowych oznacza, że obrazek *obrazek164.gif* nadal znajduje się w buforze. Inaczej mówiąc, obiekt klasy `Image` obrazka nadal jest osiągalny.

- 
- **Uwaga** • Jeżeli aplikacja będzie zwracać nieskończoną liczbę komunikatów „Rysowanie obrazka”, może to oznaczać, że przestrzeń na sterce używana przez maszynę wirtualną Javy jest większa niż przestrzeń na sterce wykorzystywana przez aktualnie używaną maszynę wirtualną, gdy aplikacja została uruchomiona w systemie Windows XP. Jeżeli przestrzeń na sterce dostępna dla maszyny wirtualnej będzie odpowiednio duża, miękkie odwołania nie będą czyszczone i aplikacja będzie mogła zwracać wyniki w nieskończoność. Aby rozwiązać ten problem, można zwiększyć rozmiar tablicy obrazków `image` (na przykład z `1024*100` do `1024*500`) i ewentualnie przypisać stałej `NUM_IMAGES` większą wartość (na przykład `500`).
- 

## Klasa `WeakReference`

Klasa `WeakReference` opisuje obiekt klasy `Reference`, którego referent jest słabo osiągalny. `WeakReference` to klasa ogólna, która dziedziczy metody po klasie `Reference`, a także udostępnia konstruktory opisane poniżej, które inicjalizują obiekty klasy `WeakReference`:

- `WeakReference(T r)` umieszcza w obiekcie odwołanie `r`. Obiekt klasy `WeakReference` zachowuje się jak słabe odwołanie do `r`. Z obiektem tym nie jest skojarzona kolejka `ReferenceQueue`.
- `WeakReference(T r, ReferenceQueue<? super T> q)` umieszcza w obiekcie odwołanie `r`. Obiekt klasy `WeakReference` zachowuje się jak słabe odwołanie do `r`. Z obiektem tym zostanie skojarzony obiekt klasy `ReferenceQueue` wskazywany przez argument `q`. Jeżeli jako `q` zostanie przekazane `null`, będzie to oznaczać, że ze słabym odwołaniem nie jest skojarzona żadna kolejka.

Klasa `WeakReference` przydaje się do zapobiegania wyciekom pamięci, które mogą wystąpić w przypadku korzystania z map skrótów. Wyciek pamięci następuje wtedy, gdy do mapy skrótów są dodawane kolejne obiekty, które nie są nigdy usuwane. Obiekty te pozostają wówczas w pamięci, ponieważ mapa skrótów przechowuje silne odwołania do nich.

Idealnym rozwiązaniem byłoby, gdyby obiekty pozostawały w pamięci jedynie wówczas, gdy w aplikacji występują silne odwołania do nich. Z chwilą zniknięcia ostatniego silnego odwołania do obiektu (nie licząc silnego odwołania w mapie skrótów) obiekt powinien zostać usunięty przez proces odśmiecania.

Okolicznościom, w których dochodzi do wycieków pamięci, można zapobiec przez tworzenie słabych odwołań do elementów mapy skrótów. Dzięki temu elementy mapy skrótów zostaną usunięte, jeśli przestaną istnieć jakiegokolwiek silne odwołania do ich kluczy. Do tego celu służy klasa `WeakHashMap` opisywana w rozdziale 8.

## Klasa PhantomReference

Klasa `PhantomReference` opisuje obiekt klasy `Reference`, którego referent jest złudnie osiągalny. `PhantomReference` jest klasą ogólną, która dziedziczy metody po klasie `Reference` i pokrywa metodę `get()`, a także udostępnia opisany poniżej konstruktor przeznaczony do inicjalizowania obiektów klasy `PhantomReference`.

- `PhantomReference(T r, ReferenceQueue<? super T> q)` umieszcza w obiekcie odwołanie `r`. Obiekt klasy `PhantomReference` zachowuje się jak złudne odwołanie do `r`. Z obiektem klasy `PhantomReference` jest skojarzony obiekt klasy `ReferenceQueue` wskazywany przez argument `q`. Przekazanie `null` do argumentu `q` nie ma sensu, ponieważ pokrywająca metoda `get()` zwraca `null` i obiekt klasy `PhantomReference` nigdy nie zostanie zakolejkowany.

W odróżnieniu od obiektów klas `WeakReference` i `SoftReference`, które są kolejkowane w ich własnych kolejkach odwołań wówczas, gdy ich referenty stają się słabo osiągalne (przed finalizacją), a czasami także po tym, gdy ich referenty stają się miękko osiągalne (przed finalizacją), obiekty klasy `PhantomReference` są kolejkowane po odzyskaniu pamięci zajmowanej przez ich referenty.

Pomimo że referent obiektu klasy `PhantomReference` jest niedostępny (metoda `get()` zwraca `null`), klasa jest jak najbardziej przydatna, ponieważ zakolejkowanie obiektu klasy `PhantomReference` wskazuje dokładnie moment usunięcia referenta. Można więc na przykład opóźnić utworzenie dużego obiektu do momentu, aż inny duży obiekt zostanie usunięty z pamięci (i uniknąć w ten sposób rzucenia obiektu klasy `java.lang.OutOfMemoryError`).

Klasa `PhantomReference` jest przydatna również dlatego, że może stanowić substytut **wskrzeszenia** (ang. *resurrection*), czyli zjawiska, w którym obiekt nieosiągalny ponownie staje się osiągalny. Jako że referent jest niedostępny (metoda `get()` zwraca `null`) ze względu na fakt, że nie ma go już w pamięci w momencie, gdy obiekt klasy `PhantomReference` jest kolejkowany, obiekt może zostać wyczyszczony w pierwszym cyklu pracy procesu odśmieciania, w którym obiekt ten zostanie zidentyfikowany jako złudnie osiągalny. Po uzyskaniu odpowiedniego powiadomienia za pośrednictwem kolejki odwołań obiektu klasy `PhantomReference` można wyczyścić wszystkie zasoby powiązane z obiektem.

- 
- **Uwaga** • Wskrzeszenie zachodzi w metodzie `finalize()`, gdy zmiennej z głównego zbioru odwołań przypisuje się `this`. W metodzie `finalize()` można na przykład wykonać instrukcję `r = this;`, aby obiekt nieosiągalny wskazywany jako `this` przypisać polu klasy o nazwie `r`.
- 

Z kolei proces odśmieciania potrzebuje co najmniej dwóch cykli, aby stwierdzić, czy obiekt pokrywający metodę `finalize()` może zostać przez ten proces przetworzony. Jeśli w pierwszym cyklu odśmieciania okaże się, że obiekt może zostać usunięty, zostanie wywołana metoda `finalize()`. Jednak metoda `finalize()` może doprowadzić do wskrzeszenia obiektu, dlatego potrzebny jest jeszcze drugi cykl, w którym proces odśmieciania sprawdzi, czy do wskrzeszenia rzeczywiście doszło.

- **Ostrzeżenie** • Zjawisko wskrzeszania wykorzystano w implementacji pul obiektów, które odtwarzają te same obiekty, jeśli ich utworzenie jest kosztowne (czasochłonne). Przykładem takiego obiektu może być choćby połączenie z bazą danych. Jednak wskrzeszanie też jest czynnością czasochłonną, a klasa `PhantomReference` likwiduje wymóg wykonywania tej czynności, a zatem zdecydowanie należy unikać wskrzeszania we własnych aplikacjach.

Na listingu 6.15 pokazano, w jaki sposób za pomocą klasy `PhantomReference` można zidentyfikować fakt usunięcia dużego obiektu.

**Listing 6.15.** *Rozpoznawanie, czy usunięty został duży obiekt*

```
class LargeObject
{
    private byte[] memory = new byte[1024*1024*50]; // 50 megabajtów
}
public class LargeObjectDemo
{
    public static void main(String[] args)
    {
        ReferenceQueue<LargeObject> rq;
        rq = new ReferenceQueue<LargeObject>();
        PhantomReference<LargeObject> pr;
        pr = new PhantomReference<LargeObject>(new LargeObject(), rq);
        int counter = 0;
        int[] x;
        while (rq.poll() == null)
        {
            System.out.println("oczekiwanie na usunięcie dużego obiektu");
            if (counter++ == 10)
                x = new int[1024*1024];
        }
        System.out.println("duży obiekt został usunięty");
    }
}
```

W kodzie na listingu 6.15 zadeklarowano klasę `LargeObject`, której prywatna tablica memory zajmuje 50 MB. Jeżeli w momencie uruchomienia aplikacji używana implementacja Javy rzuci błąd `OutOfMemoryError`, konieczne będzie zmniejszenie rozmiaru tej tablicy.

Metoda `main()` tworzy najpierw obiekt klasy `ReferenceQueue` opisujący kolejkę, do której zostanie dodany utworzony w kolejnym kroku obiekt klasy `PhantomReference` zawierający odwołanie do `LargeObject`.

Następnie metoda `main()` tworzy obiekt klasy `PhantomReference`. W tym celu do konstruktora zostaje przekazane odwołanie do nowo utworzonego obiektu klasy `LargeObject` oraz odwołanie do wcześniej utworzonego obiektu klasy `ReferenceQueue`.

Po zainicjalizowaniu zmiennej `counter` (która wskazuje liczbę iteracji, jakie wykonano przed utworzeniem kolejnego dużego obiektu) oraz po zadeklarowaniu zmiennej lokalnej o nazwie `x`, przeznaczonej do przechowywania silnego odwołania do kolejnego dużego obiektu, metoda `main()` uruchamia pętlę odpytywania.

Pętla odpytywania rozpoczyna się od wywołania metody `poll()`, która sprawdza, czy obiekt klasy `LargeObject` został usunięty z pamięci. Dopóki metoda będzie zwracać `null`, co





8. Która stała `RoundMode` opisuje tryb zaokrąglania, o którym mówi się w szkole na lekcjach?
9. Co to jest `BigInteger`?
10. Do czego służy metoda `isSealed()` klasy `Package`?
11. Metoda `getPackage()` wymaga, by z pakietu załadowano co najmniej jeden plik klas, zanim zostanie zwrócony obiekt klasy `Package` opisujący ten pakiet. Prawda czy fałsz?
12. Należy wskazać dwa najważniejsze zastosowania podstawowych klas opakowujących.
13. Dlaczego powinno się unikać implementowania wyrażen w stylu `ch >= '0' && ch <= '9'` (do sprawdzenia, czy `ch` zawiera cyfry) albo `ch >= 'A' && ch <= 'Z'` (aby sprawdzić, czy `ch` zawiera wielką literę)?
14. Należy wskazać cztery rodzaje osiągalności.
15. Co to jest referent?
16. Która z klas API `Reference` jest odpowiednikiem metody `finalize()` klasy `Object`?
17. Zanim rozpowszechniły się ekrany graficzne, czasami programiści wyświetlali kształty na ekranach tekstowych. Na przykład okrąg mógł być prezentowany w następujący sposób:

```

      *
    *****
  **          **
**            **
*              *
*              *
**            **
*              *
*              *
*              *
**            **
*              *
*              *
*              *
**            **
*              *
*              *
**            **
  **          **
    *****
      *

```

- 
- **Uwaga** • Powyższa figura ma kształt eliptyczny, a nie okrągły, ponieważ wyświetlana wysokość każdej gwiazdki jest większa niż jej wyświetlana szerokość. Gdyby wysokość i szerokość były takie same, figura miałaby kształt okręgu.
- 

Należy zaimplementować aplikację `Circle`, która będzie generować i wyświetlać przedstawiony powyżej okrąg. Najpierw aplikacja powinna tworzyć dwuwymiarową tablicę `screen` złożoną z 22 wierszy i takiej samej liczby kolumn. Każdy element tablicy trzeba zainicjalizować znakiem spacji (który wskazuje czysty ekran). Dla każdego kąta o mierze całkowitoliczbowej z przedziału od 0 do 360 należy obliczyć współrzędne  $x$  i  $y$  w ten sposób, że promień o wartości 10 będzie mnożony przez sinus i kosinus kąta. Do wartości  $x$  i  $y$  należy dodać 11, aby kształt okręgu umieścić w środku tablicy `screen`. Punktowi o wynikowych współrzędnych  $(x,y)$  należy przypisać znak gwiazdki. Gdy wykonywanie pętli dobiegnie końca, należy zwrócić zawartość tablicy do standardowego wyjścia.

18. **Liczba pierwsza** to dodatnia liczba całkowita, która jest podzielna tylko przez jeden i przez siebie samą. Należy utworzyć aplikację o nazwie `PrimeNumberTest`, która będzie sprawdzać, czy przekazany do niej pojedynczy argument całkowitoliczbowy wskazuje liczbę pierwszą, czy inną liczbę, i wyświetli stosowny komunikat. Na przykład aplikacja wywołana poleceniem `java PrimeNumberTest 289` powinna zwrócić komunikat „289 nie jest liczbą pierwszą”. Aby w prosty sposób sprawdzić, czy liczba jest liczbą pierwszą, należy przejść w pętli od liczby 2 do pierwiastka kwadratowego argumentu całkowitoliczbowego i wykorzystać w tej pętli operator reszty z dzielenia. Za jego pomocą można sprawdzić, czy argument jest podzielny przez indeks pętli. Na przykład ze względu na to, że wyrażenie  $6\%2$  ma wartość 0 (6 jest podzielne przez 2), liczba całkowita 6 nie jest liczbą pierwszą.

## Podsumowanie

Klasa `java.lang.Math` uzupełnia standardowe operacje matematyczne (realizowane przez operatory `+`, `-`, `*`, `/` oraz `%`) o możliwość wykonywania zaawansowanych operacji (na przykład obliczeń trygonometrycznych). Towarzysząca jej klasa `java.lang.StrictMath` zapewnia, że wszystkie operacje zaawansowane dadzą ten sam wynik na wszystkich platformach.

Wartości pieniężne nigdy nie powinny być reprezentowane przez wartości zmiennoprzecinkowe ani zmiennoprzecinkowe z podwójną precyzją, ponieważ nie wszystkie wartości pieniężne mają precyzyjną reprezentację. Natomiast klasa `java.math.BigDecimal` zapewnia dokładną prezentację takich wartości oraz wykonywanie na nich precyzyjnych obliczeń.

Klasa `BigDecimal` wykorzystuje klasę `java.math.BigInteger` do reprezentowania swojej wartości niewyskalowanej. Instancja klasy `BigInteger` opisuje wartość całkowitoliczbową, która może być wartością o dowolnej długości (ograniczonej jedynie limitami pamięci dostępnej dla maszyny wirtualnej).

Klasa `java.lang.Package` daje dostęp do informacji na temat pakietów. Informacje te zawierają wersję implementacji i specyfikacji pakietu języka Java, nazwę pakietu, a także wskazanie, czy pakiet jest upakowany, czy nie.

Instancje podstawowych klas opakujących `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` i `Short` z pakietu `java.lang` stanowią opakowania dla wartości typów podstawowych. Klasy te przydają się do przechowywania wartości typów podstawowych w kolekcjach.

API `References` pozwala aplikacji na interakcję — w ograniczonym zakresie — z procesem odśmiecania. Pakiet `java.lang.ref` tego API zawiera klasy `Reference`, `ReferenceQueue`, `SoftReference`, `WeakReference` oraz `PhantomReference`.

Klasa `SoftReference` przydaje się do implementowania bufora obrazków, za pomocą klasy `WeakReference` można zapobiegać wyciekowi pamięci potencjalnie zachodzącym w trakcie pracy z mapami skrótów, zaś dzięki `PhantomReference` można sprawdzać, czy obiekt został już zniszczony i czy w związku z tym można wyczyścić powiązane z nim zasoby.

Zakończyliśmy dopiero pierwszy etap poznawania podstawowych interfejsów API języka Java. W rozdziale 7. zapoznamy się z kolejnym podstawowym API — `Reflection API`, interfejsem do zarządzania ciągami znaków, klasą `System`, a także niskopoziomowym API `Threading`.

# Skorowidz

## A

- abs(), 248, 256, 261
- abstract, 58, 139, 140
- AbstractCollection, 336
- AbstractList, 336
- AbstractMap, 336
- AbstractQueue, 336
- AbstractSequentialList, 336
- AbstractSet, 336
- abstrakcyjna
  - klasa, 139
  - metoda, 140
- accept(), 470, 471
- AccessibleObject, 334
- acos(), 248
- add(), 163, 256, 261, 344
- addAppts(), 132
- adnotacja, 208, 209
  - czas życia, 215
  - Deprecated, 209
  - metody pokrywającej, 209
  - Override, 119, 209
  - przetwarzanie, 217
  - SuppressWarnings, 209, 211
  - typ, 212
  - znacznikowa, 212
- algorytm, 205
  - sortowanie bąbelkowe, 205
  - szeregowania, 311
- allOf(), 358
- alternatywa warunkowa, 66
- AND, 391
- Android, 17, 21, 56
  - aplikacja, 17
  - Platforma, 21
- annotation, 208
- annotationType, 296
- anonimowa klasa, 164, 166
  - potomna, 239
- anonymous class, 164
- API, 289, 335, 397, 415, 463, 540
  - Reference, 279
  - Reflection, 333
  - Threading, 307, 334
- aplet, 21
- aplikacja, 21, 56, 289, 307
  - dla Androida, 17
  - implementowanie, 56
  - współbieżna, 397
  - zinternacjonalizowana, 416
  - zorientowana obiektowo, 57
- append(), 524
- arcus cosinus kąta, 248
- arcus sinus kąta, 248
- arcus tangens kąta, 248
- argument, 74, 112
  - przekazywanie przez wartość, 108, 113
  - typu rzeczywistego, 221, 224
  - ukryty, 75
  - wiersza poleceń, 23
- Array, 296, 334
- arraycopy(), 304, 306, 334
- ArrayList, 336, 348
- Arrays, 387
- ASCII, 523

asercja, 200, 203, 205, 207, 244  
     włączanie, 208  
     wyłączanie, 208  
 asin(), 248  
 assert, 58, 200  
 assertion, 200  
 AssertionError, 200  
 atan(), 248  
 Atomic, 413  
 atomowy, 318  
 atrybut, 340  
     encji, 59  
     klasy, 59  
     obiektu, *Patrz* atrybut encji  
 automatyczne  
     pakowanie, 342  
 await(), 406

## B

bajt, 60  
 bariera  
     cykliczna, 406  
     wymiany, 406  
 baza danych prosta, 479  
 bazowa nazwa, 416  
 bezpieczeństwo, 327  
     typów, 219  
 biblioteka  
     kolekcji, 335, 396  
     preferencji, 444, 461  
     standardowa klas, 335  
 BigDecimal, 256  
 BigDecimal(), 256  
 BigInteger, 260  
 BigInteger(), 261  
 bin, 23  
 binarne wyszukiwanie, 388  
 binarySearch(), 388  
 bit, 60  
     wynikowy, 65, 66  
 bitowa koniunkcja, 65  
 bitowe dopełnienie, 66  
 bitowy operator, 66  
 BitSet, 356  
 BitSet, 390  
 blank final, 72  
 BlockingQueue, 408  
 blok, 76, 112

blokada, 319, 410  
     wielowejściowa, 410  
 błąd, 181, 199  
     standardowy, 23  
 boolean, 58, 60  
     accept, 471  
     accept(), 469  
     equals(), 121  
 Boolean, 267, 268, 269  
 break, 58, 81, 86  
 BreakIterator, 425, 461  
 BufferedInputStream, 504  
 BufferedOutputStream, 504  
 bufor, 504  
 byte, 58, 60, 252  
 Byte, 267  
 ByteArrayInputStream, 487, 489  
 ByteArrayOutputStream, 489

## C

Calendar, 433, 461  
 capacity, 303, 376  
 case, 58, 80  
 catch, 18, 58, 188, 189, 192, 198  
 ceil(), 248  
 CEILING, 258  
 char, 18, 58, 60, 461  
 Character, 267, 270, 461  
 charset, 528  
 charValue(), 270  
 checked exception, 184  
 CheckingAccount, 58, 90, 109  
 ciało  
     klasy, 58  
     metody, 74  
 ciąg  
     znaków, 24, 129, 297, 334  
     łączenie, 69  
 Circle, 226  
 class, 18, 20, 58, 112  
     invariant, 206  
     literal, 296  
 Class, 290, 297, 334  
 ClassCastException, 219, 245  
 classloader, 20, 173  
 ClassNotFoundException, 293  
 clear(), 279

clone(), 121, 134  
     pokrywanie, 240  
 close(), 194, 487  
 Collator, 461  
 Collection, 335, 338, 341  
 Collections, 389  
 Comparable, 227, 362  
 compare(), 337, 364  
 compareTo(), 226, 240, 268, 337, 364, 429  
 compress(), 491  
 ConcurrentLinkedQueue, 408  
 ConcurrentMap, 408  
 connect(), 174, 495  
 const, 58  
 constant interface, 179  
 Constructor, 334  
 continue, 58, 87  
 control-flow invariant, 203  
 convert(), 188  
 copy(), 195  
 cos(), 248  
 cosinus kąta, 248  
 countdown latch, 406  
 CountdownLatch, 406  
 createTempFile(), 473  
 cukierek syntaktyczny, 298  
 currentTimeMillis(), 304  
 cyclic barrier, 406  
 czas życia, 62

## D

Dalvik, 22, 56  
 data, 430  
 DataInputStream, 506  
 DataOutputStream, 506  
 Date, 430  
 deadlock, 327  
 deal(), 43  
 debugger, 27  
 Deck, 38  
 default, 58, 81  
 defekt, 199  
 deklaracja klasy, 58  
 deleteOnExit(), 473  
 deposit(), 75, 90, 109  
 deprecated, 210  
 Deprecated, 209  
 deprecation, 211

deserializacja obiektu, 508  
     domyślna, 508  
     własna, 513  
 design by contract, 203  
 developer(), 213  
 Dictionary, 390  
 disconnect(), 174  
 divide(), 256, 261  
 długość, 303  
 do, 58, 85  
 dodawanie, 65  
 domena problemu, 36  
 domknięcie, 166  
 domyślna lokalizacja, 415  
 dopasowanie  
     granic, 455  
     o zerowej długości, 455  
 dopełnienie  
     bitowe, 66, 71  
     logiczne, 67, 71  
 dostęp  
     do pola, 104, 113  
     do składowej, 68  
 double, 18, 59, 247, 254  
 Double, 60, 267, 271  
 Double(), 272  
 doubleValue(), 272  
 do-while, 81, 84  
 DOWN, 258  
 downcasting, 140  
 draw(), 138  
 drzewo preferencji  
     systemowych, 444  
     użytkownika, 444  
 dueDate(), 213  
 dynamiczny język programowania, 21  
 dziedziczenie, 19, 115, 116, 121, 156  
     implementacji, 115, 120, 130  
     wielokrotne, 120  
     interfejsu, 115, 147, 149  
     jednokrotne, 120  
     wielokrotne, 120  
 dzielenie, 67

## E

ea, 208  
 Eclipse, 27, 32, 56  
     instalacja, 32

eksternalizacja, 517  
 elastyczność kodu, 150, 156  
 element, 213, 338  
     developer(), 213  
     dueDate(), 213  
     id(), 213  
     tablicy, 61  
 else, 59, 78  
 enableassertions, 208  
 encja, 57, 112  
     abstrakcyjna, *Patrz* obiekt  
 enkapsulacja, 57, 131  
 enqueue(), 279  
 entities, *Patrz* encja  
 entry, 369  
 entrySet(), 371  
 enum, 59, 81, 235, 240, 245, 257, 356  
     rozszerzenie, 236  
 Enum, 240, 242, 245  
 enumeracja, 390  
 enumerated type, 233  
 EnumMap, 336, 383  
 EnumSet, 336, 357  
 epoka unixowa, 430  
 equals(), 125, 268, 337, 344, 364  
     pokrywanie, 240  
 erasure, 231  
 Error, 198  
 etykieta, 88, 89  
 exception, 181  
 Exception, 184, 198  
 ExceptionInInitializerError, 293  
 exchanger, 406  
 Executor, 398, 460  
 executors, 398  
 ExecutorService, 399, 460  
 exp(), 248  
 extends, 18, 59, 116, 149, 224  
 Externalizable, 517

## F

factorial(), 262  
 fail-fast, 346  
 false, 59, 63, 78, 268  
 Field, 334  
 FIFO, 365  
 File, 463

FileDescriptor, 478  
 FileFilter, 471  
 FilenameFilter, 469  
 FileOutputStream, 491  
 FileReader, 531  
 FileWriter, 529  
 FilterInputStream, 501  
 FilterOutputStream, 497  
 filtrowany strumień, 497  
     wejściowy, 501  
     wyjściowy, 497  
 final, 59, 72, 73, 116  
 finalizacja, 127  
 finalize(), 121, 127  
     pokrywanie, 240  
 finally, 59, 192, 198  
 find(), 451  
 First In, First Out, *Patrz* FIFO  
 float, 18, 59, 254  
 Float, 60, 267, 271  
 Float(), 271, 272  
 floatToIntBits(), 272  
 floatValue(), 272  
 FLOOR, 258  
 floor(), 248  
 flush(), 487  
 for, 18, 59, 81, 82  
 formal type parameter list, 221  
 format(), 255  
 formater, 436  
     daty, 439  
     komunikatu, 441  
     liczby, 436  
 Formatter, 524  
 forName(), 217, 292, 297  
 fragile base class problem, 132  
 free variable, 166  
 funkcja  
     mieszająca, 374  
     pierwszej kategorii, 166  
 Future, 401

## G

garbage collector, 110, 382  
 generator  
     liczb pseudolosowych, 446  
     liniowy kongruentny, 447

generic  
     method, 232  
     type, 221  
 generics, 218  
 get, 100  
 get(), 280, 295, 378  
 getAnnotation(), 217  
 getAnnotations(), 294  
 getBoolean(), 268  
 getBundle(), 417, 418  
 getClass(), 296, 297  
 getCause(), 183  
 getDateInstance(), 439  
 getDeclaredAnnotations(), 294  
 getDeclaringClass(), 240  
 getDefault(), 415  
 getFirstName(), 101  
 getImage(), 282  
 getImplementationTitle(), 264  
 getImplementationVersion(), 264  
 getInstance(), 437  
 getLastName(), 101  
 getMethods(), 217  
 getMessage(), 183  
 getName(), 101, 264  
 getNumberInstance(), 437  
 getObject(), 418  
 getPackage(), 264  
 getPackages(), 264  
 getPriority(), 312  
 getProperty(), 304  
 getSpecificationTitle(), 264  
 getSpecificationVendor(), 264  
 getSpecificationVersion(), 264  
 getter, 100  
 getTimeInstance(), 439  
 getTopCard(), 44  
 głębokie  
     klonowanie, 123  
     kopiowanie, 123  
 głowa kolejki, 223, 365  
 główny  
     interfejs, 335  
     plik klas, 54  
     zbiór odwołań, 277  
 Google, 17, 22, 56  
 Gosling James, 18  
 goto, 59, 88

gra kareta, 36  
 graf obiektu, 508  
 granica dopasowanie, 455  
 grep, 531  
 Groovy, 21  
 grupa przechwytywana, 454

## H

HALF\_DOWN, 258  
 HALF\_EVEN, 258  
 HALF\_UP, 258  
 hashCode(), 121, 128, 269, 344  
     pokrywanie, 240  
 HashMap, 336, 374  
 HashSet, 336, 353  
 Hashtable, 390  
 hasMoreElements(), 163  
 hasNext(), 341  
 headSet(), 359  
 heterogeniczna lista, 220  
 hierarchia klas  
     Throwables, 183  
     wyjątków, 198  
 homogeniczna lista, 220

## I

id(), 213  
 IDE, 27, 28, 32, 56  
 IdentityHashMap, 336, 380  
 identyczność, 124  
 identyfikacja typu w fazie wykonania, *Patrz* RTTI  
 identyfikator, 58  
     niezlokalizowany, 419  
 if, 18, 59, 78  
 if-else, 78, 79, 201  
 Image, 282  
 implementacja, 99, 112  
     dziedziczenie, 115, 120, 130  
     interfejsu, 145, 156  
 implementowanie  
     aplikacji, 56  
     buforów pamięci podręcznej, 281  
 implements, 59, 145  
 import, 59, 171, 180, 198  
     instrukcja, 171  
     statyczny, 179

- indefinite postponement, 312
- indeks, 25, 82
  - tablicy, 65
- Infinity, 251
- informacji ukrywanie, 99
- InheritableThreadLocal, 307, 330, 334
- InheritableThreadLocal(), 331
- inicjalizowanie pola, 62, 113
  - tablicy, 64
- inicjalizująca konstrukcja, 93
- inicjowanie leniwe, 447
- initCause(), 183
- initialValue(), 330
- inner classes, 157
- InputStream, 485, 487, 540
- InputStreamReader, 526
- instanceof, 19, 59, 126, 142
- instancja, 57
  - klasy, 57, 333
    - anonimowej, 166
  - konstrukcja inicjalizująca, 94, 98
  - obiektu, 102
  - rzucenie, 187
- instrukcja, 76, 112
  - do-while, *Patrz* do-while
  - for, *Patrz* for
  - goto, *Patrz* goto
  - if-else, 78
  - importu, 171
  - kontynuacji, 87
  - natywna, 20
  - pakietu, 171
  - prosta, 76, 112
  - przerwania, 86
    - z etykietą, 88
  - przypisania, 77
  - pusta, 76
  - switch, 80
  - while, *Patrz* while
  - złożona, 76, 112
- int, 18, 59, 60
- Integer, 267
- Integrated Development Environment, *Patrz* IDE
- interface, 59, 144, 212
- interfejs, 19, 99, 112, 115, 144, 150, 156, 179, 333
  - ciało, 144
  - Collection, 338
  - dziedziczenie, 147, 149
  - Executor, 398
  - ExecutorService, 399
  - Externalizable, 517
  - FileFilter, 471
  - FilenameFilter, 469
  - główny, 335
  - implementacja, 145, 156
  - Iterable, 338
  - List, 344
  - Lock, 410
  - Map.Entry, 372
  - nagłówek, 144
  - Queue, 365
  - Set, 351
  - SortedSet, 358
  - stałych, 179
    - wewnątrz klasy, 168
    - znacznikowy, 145
- intern(), 300
- internacjonalizacja, 414, 461
- internal invariant, 201
- interpreter, 20
- intrpretacja kodu bajtowego, 20
- invariant, 201
- is, 100
- is-a, *Patrz* relacja "jest"
- isAlive(), 313
- isAnnotationPresent(), 217
- isBlank(), 87
- isCompatibleWith(), 264
- isDigit(), 271
- isEmpty(), 38, 43
- isEnqueued(), 280
- isInfinite(), 272
- isLetter(), 271
- isLetterOrDigit(), 271
- isLowerCase(), 271
- isNaN(), 272
- ISO, 415
- isSealed(), 264
- isSorted(), 206
- isUpperCase(), 271
- isWhitespace(), 271
- Iterable, 335, 338, 341
- iteracja, 82, 83
- iterator(), 344
- iterowanie, 82



## J

jar, 23  
 Jar, *Patrz* plik JAR  
 Java, 17, 18, 22, 56, 57  
   bezpieczeństwo, 19, 21  
   EE, 21, 56  
   kompilator, 20  
   ME, 21, 56  
   Platform  
     Enterprise Edition, *Patrz* Java EE  
     Micro Edition, *Patrz* Java ME  
     Standard Edition, *Patrz* Java SE  
   przeñośność, 20  
   Runtime Environment, *Patrz* JRE  
   SE, 21, 22, 56  
     Development Kit, *Patrz* JDK  
   składnia, 18  
 java program, 23  
 java.lang, 334  
 javac, 23  
 Javadoc, 18, 23  
   komentarz, 40, 49  
   znacznik, 50  
 JDK, 22, 27, 56, 512  
   instalacja, 22  
 jednostronnie otwarty przedział, 359  
 język  
   zorientowany obiektowo, 115  
   programowania  
     dynamiczny, 21  
     zorientowany obiektowo, 57  
 JIT, 20  
 Joda Time, 436  
 join(), 313  
 jre, 23  
 JRE, 22, 56

## K

kalendarz, 430  
 kanoniczna postać pola, 378  
 kareta gra, 36  
 katalog główny, 463  
 keySet(), 371, 528  
 klasa, 19, 57, 73, 111, 333  
   abstrakcyjna, 139  
   anonimowa, 164, 166  
   instancja, 166

bazowa, 116  
   kruchość, 132  
 ciało, 58, 112  
 deklaracja, 58  
 deklarowanie, 112  
 final, 116  
 główna, 157, 198  
 implementująca, 335, 336  
 instancja, 57  
 kolekcji, 390  
 kompatybilna zmiana, 512  
 konstrukcja inicjalizująca, 93, 98  
 literał, 296, 297  
 lokalna, 166  
 moduł ładowania, 173  
 nadrzędna, 116  
 nagłówek, 58  
 narzędziowa, 254  
 nazwa, 58  
 niekompatybilna zmiana, 512  
 niezmienna, 118, 255, 334  
 opakowująca, 134  
   podstawowa, 267  
 package-private, 99  
 podrzędna, 116  
 pole, 112  
   inicjalizowanie, 62  
 pomocnicza, 335  
 potomna, 116, 131, 133, 134, 141, 143  
   anonimowa, 239  
 prywatna w ramach pakietu, 99  
 przeglądarka, 289  
 przodka, 132, 134, 141, 142, 143  
 public, 99  
 publiczna, 99  
 rodzica, 116  
 rozszerzanie, 115, 121, 134  
 składowa  
   niestatyczna, 160, 164  
   statyczna, 157  
 strumienia, 485, 486, 519, 541  
 Throwables hierarchia, 183  
 uogólniona, 218  
 wartości, *Patrz* podstawowa klasa  
   opakowująca  
 wyjątku, 184  
   użytkownika, 185  
 wywiedziona, 116  
 zagnieżdżona, 157, 169, 198

- klasa
  - zakresu znaków, 453
  - znaków, 453
    - część wspólna, 454
    - predefiniowana, 454
    - prosta, 453
    - różnica, 454
    - unia, 454
    - zanegowana, 453
- klient, 144
- klonowanie, 121
  - głębokie, 123
  - plytkie, 121, 123
- klucz, 369, 419
  - zmienny, 380
- klucz-wartość, 419, 443
- kod
  - bajtowy, 20
    - interpretacja, 20
    - weryfikator, 20
  - błędu, 181
  - klienta, 101
  - mieszający, 374, 378
  - natywne, 20
  - sprzątający
    - po rzuceniu wyjątku, 192
    - przed rzuceniem wyjątku, 194
  - uzupełnień do dwóch, 260
  - wariantu, 415
  - współbieżny, 408
  - wywołujący, 74
  - źródłowy, 18
- kodowanie
  - Huffmana, 393
  - znaków, 523
    - standard, 523
- koercja, 134, 135
- kolejka, 223, 365
  - głowa, 223, 365
  - ogon, 223, 365
  - priorytetowa, 365, 366
  - pusta, 223
  - wielopoziomowa ze sprzężeniem
    - zwrotnym, 311
- kolejność bajtów, 506
- kolekcja, 219, 335, 336, 396, 408
  - biblioteka, 335, 396
  - klasa, 390
  - konkretne klasy implementujące, 396
  - metody klasy Collections, 389
  - stała, 340
  - współbieżna, 460
- kolizja, 375
- komentarz
  - Javadoc, 40, 49
  - jednowierszowy, 59
  - wielowierszowy, 75
- komparator, 337
- kompatybilna zmiana klasy, 512
- kompilacja JIT, 20
- kompilator, 20
  - javac, 23
  - JIT, 20
    - Just In Time, *Patrz* kompilator JIT
- kompozycja, 115, 130
- komunikat
  - formater, 441
  - prosty, 440
  - złożony, 440
- koniunkcja
  - bitowa, 65
  - logiczna, 67
  - warunkowa, 66
- konkretne klasy implementujące kolekcje, 396
- konkretny typ, 221
  - parametryzowany, 222
- konstrukcja inicjalizująca, 93, 112
  - instancję, 94, 98
  - klasę, 93, 98
  - kolejność, 94
  - metodę, 98
  - pole
    - instancji, 93, 98
    - klasy, 93
- konstruktor, 91, 112
  - chroniony, 99
  - inicjalizujący obiekt klasy, 275
  - klasy
    - ArrayList, 348
    - BigDecimal, 256–57
    - BigInteger, 261
    - BitSet, 391
    - BufferedOutputStream, 504
    - ByteArrayInputStream, 490
    - ByteArrayOutputStream, 490
    - DataInputStream, 506
    - DataOutputStream, 506
    - Date \r, 431

- EnumMap, 383
  - File, 464, 465
  - FileInputStream, 492
  - HashMap, 376
  - HashSet, 353
  - InputStreamWriter, 526
  - LinkedList, 350
  - Locale, 414
  - MessageFormat, 441
  - OutputStreamWriter, 525
  - PipedInputStream, 494
  - PipedOutputStream, 494
  - PriorityQueue, 367
  - Random, 447
  - RandomAccessFile, 474
  - String, 298–300
  - StringBuffer, 301–3
  - ThreadLocal, 329
  - TreeMap, 373
  - TreeSet, 352
  - package-private, 99
  - private, 99
  - protected, 99
  - prywatny, 99
    - w ramach pakietu, 99
  - przeciążanie, 92, 112
  - public, 99
  - publiczny, 99
  - Throwables, 183
  - kontrakt, 144
  - kontrola dostępu, 99
    - poziom, 99
      - chroniony, 99
      - prywatny, 99
      - prywatny w ramach pakietu, 99
      - publiczny, 99
  - kontrolowany wyjątek, 184, 198
  - kopiowanie
    - głębokie, 123
    - płytkie, 121
  - kowariancja, 230
  - kowariantny typ, 142
  - kursor, 347, 425
    - pozycja, 347
    - wirtualny, 426
  - kwantyfikikator, 456
    - niechętny, 456, 457
    - własnościowy, 456, 457
    - zachłanny, 456
- ## L
- Last In, First Out, *Patrz* LIFO
  - leastDesirableCard(), 49
  - length, 82, 303
  - leniwe inicjowanie, 447
  - lib, 23
  - liczba
    - całkowita, 60
      - bajtowa, 60, 70
      - długa, 60, 70
      - krótka, 60, 70
    - Eulera, 248
    - pseudolosowa, 250
    - zmiennopozycyjna, 60, 70
      - o podwójnej precyzji, 60, 70
  - LIFO, 365
  - liniowe wyszukiwanie, 388
  - liniowy generator kongruentny, 447, 461
  - LinkageError, 293
  - LinkedHashMap, 336
  - LinkedHashSet, 336
  - LinkedList, 336, 349, 350
  - List, 335, 344
  - list(), 469
  - lista, 344
    - heterogeniczna, 220
    - homogeniczna, 220
    - operacja na zakresach, 348
    - parametrów, 74
      - typów formalnych, 221
    - powiązana, 350
  - ListResourceBundle, 421
  - listRoots(), 463
  - litera, 428
  - literał, 62
    - całkowitoliczbowy, 63
    - klasy, 296, 297
    - logiczny, 63
    - typ, 63
      - w postaci ciągu znaków, 63
    - zmiennopozycyjny, 64
    - znakowy, 63
  - load factor, 376
  - local class, 166
  - locale, 433
  - Locale, 414, 425, 461
  - Lock, 410
  - log(), 132, 248

log10(), 249  
 logarytm, 249  
     naturalny, 248  
 logging, 266  
 logiczna koniunkcja, 27  
 logiczne  
     dopełnienie, 67  
     wyrażenie, 83  
 logiczny operator, 67, 68  
 lokalizacja, 414, 416  
     domyślna, 415  
 lokalizator, 414  
 lokalizowanie, 416  
 lokalna klasa, 166  
 long, 18, 59  
 Long, 60, 267  
 lookingAt(), 451

## Ł

łańcuch  
     nadrzędnej paczki zasobów, 418  
     wywołań, 109, 113  
     znaków porównywanie, 429  
 łączyć, 349  
 łączenie ciągów znaków, 69  
 łączność, 70  
     operatorów  
         lewostronna, 71  
         prawostronna, 71

## M

main(), 24, 75, 122  
 manifest, 267  
 MANIFEST.MF, *Patrz* plik manifestu  
 map, 369  
 Map, 335, 382  
 Map.Entry, 372  
 mapa, 369  
     mieszająca, 378  
     uporządkowana, 384  
 mark(), 487  
 marker annotations, 212  
 markSupported(), 487  
 maskowanie zmiennej typu, 227  
 maszyna wirtualna, 20, 56  
 Matcher, 461  
 Math, 247, 461

max(), 249, 256, 261  
 MAX\_VALUE, 271  
 mechanizm ogólny, 218, 220  
 Media, 188  
 metaadnotacja, 214, 216, 245  
 meta-annotations, 214  
 metadane, 208, 245, 475  
 metaznak, 449  
 Method, 334  
 metoda, 58, 73, 112, 333  
     łańcuch wywołań, 109  
     klasy  
         Integer, 275  
         abs(), 248, 256, 261  
             przeciążanie, 252  
         abstrakcyjna, 140, 239  
         accept(), 469, 470, 471  
         acos(), 248  
         add(), 163, 256, 261, 344  
         addAppts(), 132  
         allOf(), 358  
         append(), 524  
         arraycopy(), 304, 306, 334  
         asin(), 248  
         atan(), 248  
         await(), 406  
         BigDecimal(), 256  
         BigInteger(), 261  
         binarySearch(), 388  
         ceil(), 248  
         charValue(), 270  
         chroniona, 99  
         ciało, 74  
         clear(), 279  
         clone(), 121, 134  
             pokrywanie, 240  
         close(), 194, 487  
         compare(), 337, 364  
         compareTo(), 226, 240, 268, 337, 364, 429  
         compress(), 491  
         connect(), 174, 495  
         convert(), 188  
         copy(), 195  
         cos(), 248  
         createTempFile(), 473  
         currentTimeMillis(), 304  
         deal(), 43  
         deklarowanie, 73  
         deleteOnExit(), 473

deposit(), 75, 90, 109  
 disconnect(), 174  
 divide(), 256, 261  
 dopasowująca, 449  
 doubleValue(), 272  
 draw(), 138  
 enqueue(), 279  
 entrySet(), 371  
 equals(), 121, 125, 268, 337, 344, 364  
     pokrywanie, 240  
 exp(), 248  
 fabrykująca klasy  
     BreakIterator, 425  
     DateFormat, 439  
 factorial(), 262  
 finalize(), 121, 127  
     pokrywanie, 240  
 find(), 451  
 floatToIntBits(), 272  
 floatValue(), 272  
 floor(), 248  
 flush(), 487  
 format(), 255  
 forName(), 217, 292, 297  
 get(), 280, 295, 378  
 getAnnotation(), 217  
 getAnnotations(), 294  
 getBoolean(), 268  
 getBundle(), 417, 418  
 getClass(), 121, 296, 297  
 getDateInstance(), 439  
 getDeclaredAnnotations(), 294  
 getDeclaringClass(), 240  
 getDefault(), 415  
 getFirstName(), 101  
 getImage(), 282  
 getImplementationTitle(), 264  
 getImplementationVendor(), 264  
 getImplementationVersion(), 264  
 getInstance(), 437  
 getLastName(), 101  
 getMethods(), 217  
 getName(), 101, 264  
 getNumberInstance(), 437  
 getObject(), 418  
 getPackage(), 264  
 getPackages(), 264  
 getPriority(), 312  
 getProperty(), 304  
 getSpecificationTitle(), 264  
 getSpecificationVendor(), 264  
 getSpecificationVersion(), 264  
 getTimeInstance(), 439  
 getTopCard(), 44  
 hashCode(), 128, 269, 344  
     pokrywanie, 240  
 hasMoreElements(), 163  
 hasNext(), 341  
 headSet(), 359  
 initialValue(), 330  
 instancji, 75  
 int hashCode(), 121  
 interfejsu  
     Collection, 338–40  
     ExecutorService, 399–401  
     Externalizable, 517  
     Future, 402  
     list, 344–46  
     ListIterator, 346–47  
     Map, 371  
     Map.Entry, 373  
     Queue, 366  
     SortedMap, 386  
     SortedSet, 360  
 intern(), 300  
 isAlive(), 313  
 isAnnotationPresent(), 217  
 isBlank(), 87  
 isCompatibleWith(), 264  
 isDigit(), 271  
 isEmpty(), 38, 43  
 isEnqueued(), 280  
 isInfinite(), 272  
 isLetter(), 271  
 isLetterOrDigit(), 271  
 isLowerCase(), 271  
 isNaN(), 272  
 isNegative(), 269  
 isSealed(), 264  
 isSorted(), 206  
 isUpperCase(), 271  
 isWhitespace(), 271  
 iterator(), 344  
 join(), 313  
 keySet(), 371, 528

metoda

klasy, 75

- Arrays, 388
- BigDecimal, 256–57
- BigInteger, 261
- BitSet, 391
- Boolean, 268, 269
- BreakIterator, 425, 426
- Calendar, 433
- Collator, 429
- Collections, 389
- Date, 431
- File, 466, 468, 470, 473, 475
- FileDescriptor, 478
- InputStream, 488
- InputStream, 489
- Math, 248–50
- OutputStream, 488
- Package, 264
- Pattern, 450
- PatternSyntaxException, 451
- PipedInputStream, 495
- PipedOutputStream, 494
- Random, 448
- RandomAccessFile, 478
- Reference, 279
- ReferenceQueue, 280
- String, 298–300
- StringBuffer, 301–3
- System, 305
- Thread, 308–9
- ThreadLocal, 329
- AccessibleObject, 296
- Class, 290–92
- Constructor, 294–95
- Field, 295
- Method, 295–96
- Object, 121
- konstrukcja inicjalizująca, 98
- leastDesirableCard(), 49
- list(), 469
- listRoots(), 463
- log(), 132, 174, 248
- log10(), 249
- lookingAt(), 451
- main(), 24, 75, 122
- mark(), 487
- markSupported(), 487

- max(), 249, 256, 261
  - przeciążanie, 252
- min(), 249, 256, 261
  - przeciążanie, 252
- multiply(), 257, 261
- name(), 241
- narzędziowa klasy
  - Character, 270
  - Double, 272
  - Float, 272
  - Preferences, 444
- natywna, 504
- nazwa, 73
- negate(), 257, 261
- newLogger(), 175
- next(), 341, 347
- nextElement(), 163
- nextIndex(), 347
- niestatyczna, 232
- notify(), 121, 323
- notifyAll(), 121
- object clone(), 121
- odczytująca, 100
- of(), 358
- offer(), 365
- ogólna, 232
  - parametr typu, 233
- ordinal(), 49, 241
- outputList(), 228
- package-private, 99
- parseBoolean(), 269
- parseFloat(), 272
- pokrywająca, 142
  - adnotacja, 209
- pokrywanie, 118, 124, 130, 132
- pomocnicza, 99
- precision(), 257
- previous(), 347
- previousIndex(), 347
- print(), 520
- printBalance(), 73, 77, 109
- println(), 24, 520
- printReport(), 109
- printStackTrace(), 183
- private, 99
- protected, 99
- prywatna, 99
- prywatna w ramach pakietu, 99

- przeciążanie, 91, 112, 119
  - przeciążona, 469
  - przekazująca, 133
  - public, 99
  - publiczna, 99
  - put(), 378
  - putBack(), 43
  - random(), 249
  - range(), 358
  - rank(), 41
  - read(), 501, 504
  - readLine(), 85
  - ready(), 525
  - remainder(), 257, 261
  - remove(), 341, 344
  - reset(), 487
  - rnd(), 251
  - round(), 249
  - run(), 307, 315
  - scale(), 257
  - set(), 433
  - setFirstName(), 101
  - setLastName(), 101
  - setName(), 101
  - setScale(), 257
  - setText(), 427
  - setTopCard(), 44
  - showUsage(), 83
  - shuffle(), 43, 448
  - signum(), 249
  - sin(), 250
  - sortowania, 364
  - split(), 217
  - sqrt(), 250
  - StackTraceElement[] getStackTrace(), 183
  - statyczna, 232
  - stos wywołań, 107, 113
  - submit(), 402
  - subSet(), 359
  - subtract(), 257, 261
  - sygnatura, 74
  - T childValue(T parentValue), 331
  - tailSet(), 359
  - tan(), 250
  - Throwable getCause(), 183
  - Throwable initCause(Throwable cause), 183
  - Throwable(), 183
  - Throwables, 183
  - toDegrees(), 250, 253
  - toLowerCase(), 271
  - topCard(), 44
  - toRadians(), 250, 253
  - toString(), 121, 129, 132, 238, 257, 261, 269, 301
    - pokrywanie, 240
  - toUpperCase(), 271
  - ustawiająca, 100
  - valueOf(), 241, 269
  - values(), 371
  - void, 89
  - wait(), 121, 323
  - withdraw(), 75
  - write(), 498, 504, 525
  - wyjście, 89
  - wywołanie, 62, 68, 77, 106, 113
    - zwracanie wartości, 90
  - miara kąta przekształcenie, 250
  - MIDlet, 21
  - mieszająca funkcja, 374
  - mieszający kod, 374
  - mieszanie, 128
  - miękkie odwołanie, 278
  - min(), 249, 256, 261
  - MIN\_VALUE, 271
  - minus unarny, 70, 71
  - mnożenie, 68
  - moduł, 178
    - ładowania klas, 173
  - monitor, 319
  - multiply(), 257, 261
- ## N
- nadrzędna ścieżka, 465
  - nagłówek klasy, 58
  - najmniejsza wartość, 248
  - największa wartość, 248, 249
  - name(), 241
  - namespaces, 169
  - narzędzia wspomagające współbieżność, 397
  - native, 59
  - naturalne porządkowanie obiektów, 226
  - natywna instrukcja, 20
  - natywny kod, 20
  - Naughton Patrick, 18
  - nazwa
    - bazowa, 416
    - klasy, 58
    - rodziny, 416

- negate(), 257, 261
  - NEGATIVE\_INFINITY, 271
  - nested classes, 157
  - NetBeans, 27, 28, 56
    - instalacja, 28
    - obszar
      - edytora, 31
      - nawigatora, 31
      - projektu, 30
      - zadań, 31
  - new, 59, 102, 113
  - newLogger(), 175
  - next(), 341, 347
  - nextElement(), 163
  - nextInt(), 347
  - niedomknięty przedział, 362
  - niekompatybilna zmiana klasy, 512
  - niekontrolowany wyjątek, 184
  - nieosiągalny obiekt, 278
  - nierówność, 67
  - niestaticzna klasa składowa, 160, 164
  - niezmiennik, 201, 203
    - klas, 206
      - sterowania przebiegiem wykonania, 244
      - sterujący przebiegiem wykonania, 203
      - wewnętrzny, 201, 244
  - niszczenie obiektów, 110, 113
  - nonstatic member class, 160
  - normalizowanie ścieżki, 465
  - notify(), 121, 323
  - notifyAll(), 121, 323
  - null, 59
  - NumberFormat, 255, 438
- O**
- Oak, 18
  - obiekt, 19, 57, 73, 110, 111, 115
    - błędu, 182
    - Deck, 38
    - deserializacja, 508
    - instancja, 102
    - naturalne porządkowanie, 226
    - nieosiągalny, 278
    - niszczenie, 110
    - odwołanie, 102
    - osiągalny, 278
      - miętko, 278
      - silnie, 278
      - słabo, 278
      - złudnie, 278
    - rzucany, 182
    - serializacja, 508
    - tworzenie, 68, 102
    - wyjątku, 182
  - Object, 121, 296
    - metody, 121
  - object clone(), 121
  - ObjectOutputStream, 509
  - obsługa wyjątku, 188, 198
    - rzucanego, 188
  - obszar
    - edytora, 31
    - nawigatora, 31
    - projektu, 30
    - zadań, 31
  - obustronnie domknięty przedział, 362
  - odejmowanie, 69
  - odraczanie w nieskończoność, 312
  - odśmiecianie proces, 278
  - odśmiecianie, 110, 113
  - odwołanie, 102
    - do obiektu, 113
    - miękkie, 278
    - silne, 278
    - słabe, 278
    - złudne, 279
  - of(), 358
  - offer(), 365
  - ogon kolejki, 223, 365
  - ogólna metoda, 232
  - ograniczenie
    - argumentów typu rzeczywistego, 224
      - dolne, 226
      - górne, 224
    - typu rekurencyjne, 227
  - ONE, 260
  - opakowująca klasa, 134
  - opakowywanie wartości typu, 342
  - operacja
    - wejścia-wyjścia, 487, 504, 524, 540
    - na zakresach listy, 348
  - operand, 65, 66, 67, 68, 69, 70
  - operator, 62, 65, 66
    - alternatywa warunkowa, 66
    - arytmetyczny, 18
    - binarny, 65
    - bitowa koniunkcja, 65



- bitowe dopełnienie, 66
- bitowy, 66
- dodawanie, 65
- dopełnienie
  - logiczne, 71
  - bitowe, 71
- dostęp do składowej, 68
- dostępu, 105, 113
- dzielenie, 67
- indeks tablicy, 65
- koniunkcja
  - logiczna, 67
  - warunkowa, 66
- logiczna koniunkcja, 67
- logiczne dopełnienie, 67
- logiczny, 67, 68
- łączenie ciągów znaków, 69
- łączność
  - lewostronna, 71
  - prawostronna, 71
- mnożenie, 68
- new, 102, 113
- nierówność, 67
- odejmowanie, 69
- postdekrementacja, 68
- postinkrementacja, 68
- predekrementacja, 68, 71
- preinkrementacja, 68, 71
- priorytet, 70
- przeciążanie, 19
- przedrostkowy, 65
- przesunięcia, 19
  - w lewo, 67
  - w prawo bez znaku, 70
  - w prawo ze znakiem, 69
- przypisanie, 65, 71
  - złożone, 66, 71
- przyrostkowy, 65
- relacyjne sprawdzenie typu, 69
- relacyjny, 68, 69
- reszta z dzielenia, 69
- równość, 67
- rzutowanie, 66, 71
- trójskładnikowy, 65
- tworzenie obiektu, 68, 71
- unarny, 65
  - minus, 70, 71
  - plus, 70, 71
- warunek, 66
- warunkowa
  - alternatywa, 66
  - koniunkcja, 66
- warunkowy, 18, 71
- włączający, 66, 68
- wyłączający, 66, 67
- wywołanie metody, 68
- wzrostkowy, 65
- opróżnienie zawartości buforów, 478
- OR, 391
- Oracle, 18
- ordinal(), 49, 241
- osiągalny obiekt, 278
- outputList(), 228
- OutputStream, 485, 487, 491, 540
- OutputStreamWriter, 525
- Override, 209

## P

- package, 59, 169, 171
- package-private, 69
- paczka
  - właściwości, 419
  - zasobów, 416
    - w formie listy, 421
  - nadrzędna łańcuch, 418
- pakiet, 169, 171, 178, 198, 263
  - bez nazwy, 171
  - instrukcja, 171
  - logging, 266
  - nazwa, 170
  - podrzędny, 171
  - upakowany, 263
- pakowanie automatyczne, 342
- parameterized type, 220
- parametr, 74, 112, 135
  - typu, 221, 222
    - metody ogólnej, 233
    - nieograniczony, 224
    - ograniczony, 224
    - widoczność, 226
    - zasięg, 226
- parametryczność, 135
- parent, 465
- parseBoolean(), 269
- parseFloat(), 272

- parsowanie, 269
- partycja, 469
- Pattern, 449, 461
- pętla, 76, 81, 86
  - do-while, 81, 84
  - for, 81, 82
  - nieskończona, 86
    - przerwanie, 86
  - while, 19, 81, 87, 341
  - zagnieżdżona, 88
  - zmienna sterująca, 82
- PhantomReference, 279, 284
- pierwiastek kwadratowy, 250
- PipedInputStream, 494
- PipedOutputStream, 494
- platforma, 17, 19, 20, 56
  - Android, 21
  - kolekcji, 268
    - języka Java, 219
- plik
  - binarny, 492
  - JAR, 54, 178, 263, 267
  - Java ARchive, 54
  - klasy, 20
    - główny, 54
  - manifestu, 54
  - tymczasowy, 473
  - właściwości, 419, 443
  - wyjściowy, 25
- plus unarny, 70, 71
- płytkie
  - klonowanie, 121, 123
  - kopiowanie, 121
- początkowa pojemność, 348
- podrzędny typ, 135
- podstawowa klasa opakowująca, 267
  - Character, 270
- Point, 230
- pojemność, 303, 376
  - początkowa, 348
- pokrywanie metody, 118, 124, 130, 132
- pole, 58, 59, 112, 333, 479
  - chronione, 99
  - deklarowanie, 59
  - dostęp, 104, 113
  - finalne puste, 72
  - identyfikator, 59
  - instancji, 61, 105, 112
    - konstrukcja inicjalizująca, 93, 98
    - klasy, 62, 112
      - inicjalizowanie, 62
      - konstrukcja inicjalizująca, 93
    - package-private, 99
    - postać kanoniczna, 378
    - poziom dostępu, 105
    - private, 99, 105
    - protected, 99
    - prywatne, 99, 105, 117
      - w ramach pakietu, 99
    - public, 99, 105
    - publiczne, 99, 105
    - tablicowe, 61
    - tylko do odczytu, 72
    - typ, 59
    - wartość, 62
      - zaciemnianie, 105, 113
- polimorfizm, *Patrz* wielopostaciowość
- porównywanie
  - łańcuchów znaków, 429
  - tekstów, 429
- porządek, 336
  - alfabetyczny, 336
  - leksykograficzny, 336
  - naturalny, 336
  - słownikowy, *Patrz* porządek alfabetyczny
- porządkowa wartość, 241
- POSITIVE\_INFINITY, 271
- posortowana mapa, 528
- postcondition, 205
- postdekrementacja, 68
- postinkrementacja, 68
- poziom dostępu, 105, 112
  - chroniony, 112
  - prywatny, 112
  - prywatny w ramach pakietu, 112
  - publiczny, 112
- pozycja kursora, 347
- późne wiązanie, 138
- precision(), 257
- precondition, 204
- precyzja, 255
- predefiniowana klasa znaków, 454
- predekrementacja, 68, 71
- preemptive scheduling, 311
- preferencja, 443, 461
  - biblioteka, 444
- preinkrementacja, 68, 71

previous(), 347  
 previousIndex(), 347  
 primitive wrapper classes, 267  
 print(), 520  
 printBalance(), 73, 77, 109  
 println(), 24, 520  
 printReport(), 109  
 PrintStream, 519, 522  
 priority, 312  
 PriorityQueue, 336, 366, 367  
 priorytet, 311, 312  
     operatora, 70  
 private, 59, 99, 112  
 problem  
     bazowy, 108  
     kruczości klasy bazowej, 132  
     wiszącego else, 79  
 profilowanie kodu, 151  
 projektowanie zgodnie z kontraktem, 203, 244  
 Properties, 390, 443  
 property, 419  
 protected, 59, 99, 112, 122  
 przechwytywana grupa, 454  
 przeciążanie, 135  
     konstruktora, 92, 112  
     metody, 91, 112, 119  
 przedział  
     czasowy, 311  
     jednostronnie otwarty, 359  
     niedomknięty, 362  
     obustronnie domknięty, 362  
 przeglądarka klasy, 289  
 przekazująca metoda, 133  
 przekazywanie, 133  
     przez wartość, 108  
 przenośność, 20  
 przerwanie, 86  
 przestarzały element, 209  
 przestrzeń nazw, 169  
 przesunięcie  
     w lewo, 67  
     w prawo  
         bez znaku, 70  
         ze znakiem, 69  
 przetwarzanie adnotacji, 217  
 przodek, 121  
 przyczyna, 183  
 przypisanie, 65, 71  
     złożone, 66, 71

pseudokod, 36, 56  
     przekształcenie na kod, 38  
 public, 18, 59, 99, 112  
 pula wątków, 397  
 punkt kodowy, 523  
 puste pole finalne, 72  
 put(), 378  
 putBack(), 43

## Q

queue, 223, 365  
 Queue, 223, 335, 365

## R

race condition, 318  
 Random, 447, 461  
 random(), 249, 251, 461  
 RandomAccessFile, 474, 476, 479, 540  
 range(), 358  
 rank(), 41  
 read(), 501, 504  
 Reader, 524, 525  
 readLine(), 85  
 ready(), 525  
 ReentrantLock, 410  
 refaktoryzacja, 87  
 Reference, 279  
 ReferenceQueue, 279, 280  
 referencja wsteczna, 455  
 Reflection API, 289  
 refleksja, 142, 289, 333  
 regex, 449  
 regexp, 449  
 reification, 230  
 rejestrator, 174  
 rekord, 479  
 rekurencja, 107, 113  
 rekurencyjne ograniczenie typu, 227  
 relacja „jest”, 116  
 relacyjne sprawdzenie typu, 69  
 relacyjny operator, 68, 69  
 remainder(), 257, 261  
 remove(), 341, 344  
 reset(), 487  
 ResourceBundle, 418  
 reszta z dzielenia, 69  
 Retention, 215

return, 59, 89, 90  
     this, 109  
 rnd(), 251  
 rodziny nazwa, 416  
 root set of references, 277  
 round robin scheduling, 311  
 round(), 249  
 RoundingMode, 257  
 rozgłaszanie zdarzeń, 423  
 rozpakowywanie, 343  
 rozszerzanie klasy, 115, 121, 134  
 równość, 67  
 RTTI, 141, 289  
 run(), 307, 315  
 Runnable, 307, 334  
 runtime type identification, *Patrz* RTTI  
 RuntimeException, 184, 198  
 rzucany obiekt, 182  
 rzucenie  
     instancji, 187  
     wyjątku, 186, 198, 315  
         kod sprzątający po, 192  
         kod sprzątający przed, 194  
         ostatnie, 191  
 rzutowanie, 66, 71, 252  
     w dół, 140, 143  
     w górę, 136, 143

## S

scale(), 257  
 ScheduledExecutorService, 460  
 ScrambledInputStream, 501  
 ScrambledOutputStream, 498  
 sealed(), 263  
 sekcja  
     inicjalizacyjna, 82  
     test, 82  
     uaktualnienia, 82  
 sekcja krytyczna, 319  
 sekwencja, 344  
     ucieczki, 63  
         Unicode, 63  
 sekwencja znaków, 334  
 selektor  
     wyrażenie, 80  
 semafor, 406  
 semaphore, 406

separator, 49, 58, 464, 465  
 Serializable, 513  
 serializacja obiektu, 508  
     domyślna, 508  
     nieograniczona, 508  
     własna, 513  
 serialver, 512  
 Servlet API, 21  
 serwlet, 21  
 set, 100, 351  
 Set, 335, 357, 364  
 set(), 433  
 setFirstName(), 101  
 setLastName(), 101  
 setName(), 101  
 setScale(), 257  
 setter, 100  
 setText(), 427  
 setTopCard(), 44  
 Shape, 139  
 Sheridan Mike, 18  
 short, 18, 59, 60, 252  
 Short, 267  
 showUsage(), 83  
 shuffle(), 43, 448  
 signum(), 249  
 silne odwołanie, 278  
 silnia, 107  
 sin(), 250  
 sinus kąta, 250  
 skala, 255  
 składnia, 18, 63  
     ucieczki, 63  
 składowa dostęp, 68  
 skrót, 66  
 słabe odwołanie, 278  
 słowo zastrzeżone, 58  
 SoftReference, 279, 280  
 SortedMap, 335, 384  
 SortedSet, 335, 358, 364  
 sortowanie, 364  
     bąbelkowe, 205  
 specyfikator formatu, 521  
 split(), 217, 301  
 sprawdzanie idyntityczności, 124  
 sprzątający kod, 192, 194  
 sqrt(), 250  
 Stack, 390

StackTraceElement[] getStackTrace(), 183  
 stała, 73
 

- fazy kompilacji, 73
- typu RoundingMode, 258

 standardowa biblioteka klas, 20, 335
 

- standardowe
  - wejście, 23, 25, 334
  - wejście-wyjście, 23
  - wyjście, 23, 25, 334

 standardowy błąd, 23, 334
 

- starvation, 312
- static, 59, 73, 93, 112, 157
- statyczna klasa składowa, 157
- sterta, 102, 113
- stos, 365
  - wyjątków, 315
  - wywołań metod, 107, 113
- strefa czasowa, 430
- strictfp, 18, 59, 253
- StrictMath, 253
- String, 59, 297, 301, 336, 429
- StringBuffer, 125, 301
- struktura danych, 57, 223
- strumień, 485
  - filtrowany, 497, 501
  - klasa, 485, 486
  - wejściowy, 485
    - dekompresujący dane, 486
    - filtrowany, 501
  - wyjściowy, 485
    - filtrowany, 497
    - kompresujący dane, 486
- Stub, 215
- submit(), 402
- Subset, 160
- subSet(), 359
- subtract(), 257, 261
- Sun Microsystems, 17, 18
- super, 59, 117
- super(), 117
- SuppressWarnings, 209, 211
- Swing, 423
- switch, 18, 59, 80, 202
- swobodny dostęp, 474
- sygnatura metody, 74
- symbol wieloznaczny, 222, 228
- synchronizacja, 319, 320, 327, 410

- synchronizator, 406, 460
  - bariera
    - cykliczna, 406
    - wymiany, 406
  - semafor, 406
    - zatrząsk zliczający w dół, 406
- synchronized, 18, 59, 410
- syntetyczna zmienna, 165
- synthetic variable, 165
- System, 304
- System.err, 25
- System.out, 24, 25
- system plików, 463
- szeregowania algorytm, 311
- szeregowanie
  - karuzelowe, 311
  - wyprzedzające, 311

## Ś

- ścieżka, 464
  - bezwzględna, 465
  - nadrzędna, 465
  - nazwa, 464
  - normalizowanie, 465
  - względna, 465
- środowisko
  - leksykalne, 166
  - programistyczne, 56
    - Eclipse, 56, *Patrz Eclipse*
    - IDE, 56, *Patrz IDE*
    - JDK, *Patrz JDK*
    - NetBeans, 56, *Patrz NetBeans*
  - wykonawcze, 20, 56

## T

- T childValue(T parentValue), 331
- tablica, 61, 82
  - dwuwymiarowa, 61, 104, 113
    - inicjalizacja, 104
  - element, 61
  - implementacja, 103
  - indeks, 65, 82
  - inicjalizowanie pól, 64
  - jednowymiarowa, 61, 113
  - kowariancja, 230
  - metoda klasy Arrays, 387

- tablica
    - tworzenie, 103
    - uściślanie, 230
    - wartości, 113
    - wyszukiwanie, 388
  - tablicowy typ, 222
  - tailSet(), 359
  - tan(), 250
  - tangens kąta, 250
  - Target, 214
  - TEN, 260
  - this, 59, 93, 105, 112, 113, 164
  - Thread, 307, 312, 334
  - ThreadGroup, 307, 334
  - Threading, 307
  - ThreadLocal, 307, 329, 334
  - threads, 307
  - throw, 59, 186, 187, 198
  - Throwable, 198
  - Throwable(), 183
  - throwables, 182
  - throws, 59, 186, 187, 189, 198
  - toDegrees(), 250, 253
  - token, 238, 269, 319
  - Token, 238
  - toLowerCase(), 271
  - topCard(), 44
  - top-level classes, 157
  - toRadians(), 250, 253
  - toString(), 121, 129, 132, 238, 257, 261, 269, 301
    - pokrywanie, 240
  - toUpperCase(), 271
  - transient, 18, 59, 510
  - TreeMap, 336, 373, 386
  - TreeSet, 336, 351, 352
  - true, 59, 63, 78
  - TRUE, 268
  - try, 18, 59, 188, 189, 192, 198
  - tworzenie obiektu, 68, 71, 102
  - typ, 58
    - abstrakcyjny, 152
    - adnotacji, 212, 333
      - Retention, 215
      - Stub, 215
      - Target, 214
    - całkowitoliczbowy, 70
    - char, 60
    - konkretny, 221
    - kowariantny, 142
    - liczbowy, 60, 70
    - literału, 63
    - logiczny, 60
    - ogólny, 221, 245
      - surowy, 222
      - własny, 222
    - ograniczenie rekurencyjne, 227
    - parametryzowany, 220, 221, 227
      - konkretny, 222
    - podrzędny, 135, 228
    - podstawowy, 60
    - poła, 59, 62
    - surowy, 222, 228
    - tablicowy, 222
    - wyliczeniowy, 81, 233, 234, 245, 333
      - enum, *Patrz* enum
    - wynikowy, 74
    - znakowy, 60
- U**
- uchwyt, 102, 113, 182, 478
  - ukrywanie
    - implementacji, 102
    - informacji, 99, 112
  - unarny
    - minus, 70, 71
    - plus, 70, 71
  - unchecked, 211
    - exception, 184
  - unia klas znaków, 454
  - Unicode, 60, 414, 429, 461, 523
  - unikalny identyfikator strumienia, 512
  - unnamed package, 171
  - UNNECESSARY, 258
  - UP, 258
  - upakowany pakiet, 263
  - upcasting, 136
  - uporządkowana mapa, 384
  - uporządkowany zbiór, 358
  - ustawienia konfiguracyjne, 443
  - usuwanie, 231
    - nieużytków, 382
  - uściślanie, 230
    - tablica, 230
  - UTF, 523

## V

valueOf(), 241, 269  
 values(), 371  
 variant, 415  
 Vector, 390  
 void, 59, 74, 89  
 volatile, 59, 323

## W

wait(), 121, 323  
 wartość  
   bezwzględna, 248, 250  
   pola, 62  
   porządkowa, 241  
   specjalna, 251  
   zlokalizowana, 419  
 warunek, 66, 323  
   końcowy, 205  
   początkowy, 204  
   wyścigu, 318  
 warunkowa  
   alternatywa, 66  
   koniunkcja, 66  
 warunkowy operator, 71  
 wątek, 307, 320, 334  
   potomka, 330  
   pula, 397  
   rodzica, 330  
   synchronizacja, 319  
   zmienna lokalna, 329  
 wczesne wiązanie, 138  
 WeakHashMap, 336, 382  
 WeakReference, 279, 283  
 weryfikator kodu bajtowego, 20  
 węzeł, 349  
   inode, 393  
   klasy Preferences, 444  
 węzły powiązane, 349  
 while, 18, 19, 59, 81, 83, 85, 87, 341  
 wiązanie  
   późne, 138  
   wczesne, 138  
 widoczność parametrów typu, 226  
 widok, 348  
 wielokrotne dziedziczenie implementacji, 120  
 wielopostaciowość, 19, 115, 134, 135, 156, 227  
 wieloznaczny symbol, 222

wiersz poleceń, 23  
   argument, 23  
 wirtualny  
   system plików, 463  
   kursor, 426  
 withdraw(), 75  
 własny typ ogólny, 222  
 właściwość, 340  
 włączający operator, 66, 68  
 włączanie asercji, 245  
 wolna zmienna, 166  
 wpis, 369  
 wrapper class, 134  
 write(), 498, 504, 525  
 Writer, 524  
 wskaźnik, 19  
   pliku, 476  
 wskrzeszenie, 284  
 współbieżna aplikacja, 397  
 współbieżnie wykonywane wątki, 311  
 współbieżność, 460  
   narzędzia wspomagające, 397  
 współbieżny kod, 408  
 współczynnik zapełnienia, 376  
 wsteczna referencja, 455  
 wyciek pamięci, 110, 113  
 wyjątek, 180, 181, 198  
   fazy wykonania, 184  
   kontrolowany, 184, 189, 198  
   niekontrolowany, 184, 198  
   rzucenie, 198  
 wykonawca, 398  
 wyłączający operator, 66, 67  
 wyrażenie, 62  
   logiczne, 83  
   proste, 62, 65  
   regularne, 449  
   złożone, 65  
 wyszukiwanie  
   binarne, 388  
   liniowe, 388  
   w fazie  
   kompilacji, 173  
   wykonania, 173  
 wyścigu warunek, 318  
 wywołanie  
   metody, 62, 68, 77, 106, 113  
   rekurencyjne, 107  
   zwrotne, 134

wzajemne wykluczanie, 319  
wzorzec, 441, 449, 461  
    projektowy Dekorator, 134

## X

Xlet, 21  
XOR, 391

## Z

zaciemnianie pola, 105, 113  
zadanie, 398  
zagłodzenie wątku, 312  
zakleszczenie, 327  
zakończenie wiersza, 453  
zaokrąglenie, 258  
    liczby, 249  
zapełnienia współczynnik, 376  
zarządzanie zasobami automatyczne, 196  
zasięg, 62  
    parametrów typu, 226  
zatrząsk zliczający w dół, 406  
zbiór, 351  
    bitowy, 356  
    uporządkowany, 358  
zdarzenie rozgłaszanie, 423  
ZERO, 260

zestaw znaków, 523, 528  
    automatycznie wykrywany, 528  
ziarno, 447  
zintegrowane środowisko programistyczne IDE,  
    *Patrz* IDE  
złożone przypisanie, 66  
złudne odwołanie, 279  
zmienna, 59, 62, 112  
    atomowa, 413, 460  
    lokalna, 76  
        deklaracja, 76  
        wątku, 329  
    nazwa, 76  
    odwołania, 103  
    sterująca pętlą, 82  
    syntetyczna, 165  
    wielowartościowa, 61  
    wolna, 166  
zmienny klucz, 380  
znacznik Javadoc, 50  
znak  
    dopasowania granic, 455  
    liczby, 249, 250  
    separatora, 464  
zone, 433

## Ż

żywołność, 327



# 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**

**J**ęzyk Java, obecny na rynku od dobrych piętnastu lat, ugruntował już swoją pozycję w środowisku programistów. Wykorzystywany jest niemal w każdej dziedzinie informatycznego świata, od aplikacji internetowych, poprzez tradycyjne oprogramowanie biurowe, aż po rozwiązania dla urządzeń przenośnych. Został on również wybrany jako główny język platformy Android — aplikacje dla tego systemu pisane są w języku Java i korzystają z różnorodnych standardowych API tego języka. Dzięki temu tysiące programistów niemal z marszu rozpoczęło tworzenie aplikacji przeznaczonych dla systemu Android, właściwie nie ponosząc żadnych dodatkowych kosztów.

Jedną z najlepszych książek wprowadzających do języka Java... trzymasz właśnie w rękach. Dzięki niej błyskawicznie opanujesz język Java oraz zasady programowania obiektowego, skupiając się przede wszystkim na tych aspektach Javy, które pozwolą Ci zrozumieć istotę tworzenia aplikacji. Z tak solidnymi fundamentami zaczniesz odkrywać możliwości i ograniczenia Javy. Na kolejnych stronach znajdziesz szczegółowo omówione API platformy, jej potencjał w zakresie korzystania z kolekcji oraz tworzenia aplikacji wielojęzycznych. Ponadto nauczysz się wykonywać operacje wejścia-wyjścia. Książka ta jest idealną pozycją dla wszystkich osób chcących poznać niuanse języka Java, a następnie wykorzystać je podczas tworzenia aplikacji. Dzięki zawartym w niej ćwiczeniom błyskawicznie zweryfikujesz zdobytą wiedzę.

- **Java — język programowania i platforma**
- **Instalacja i korzystanie ze środowiska programistycznego Eclipse i NetBeans**
- **Elementarz języka JAVA — klasy, interfejsy, dziedziczenie**
- **Zaawansowane elementy języka — wyjątki, adnotacje, typy ogólne i wyczerpieniowe**
- **Zastosowanie wątków**
- **Zarządzanie kolekcjami elementów — Collections Framework**
- **Operacje wejścia-wyjścia — dostęp do plików i strumieni**

**Wykorzystaj potencjał Javy i przygotuj się do tworzenia aplikacji na urządzenia przenośne!**

**helion.pl**  
księgarnia internetowa

Nr katalogowy: 7115

 **Księgarnia internetowa**  
<http://helion.pl>

 **Zamówienia telefoniczne:**  
**0 801 339900**  
 **0 601 339900**



**Helion**

Sprawdź najnowsze promocje:  
• <http://helion.pl/promocje>  
Książki najchętniej czytane:  
• <http://helion.pl/bestsellery>  
Zamów informacje o nowościach:  
• <http://helion.pl/nowosci>

Helion SA  
ul. Kościuszki 1c, 44-100 Gliwice  
tel.: 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
<http://helion.pl>

sięgnij po **WIĘCEJ**



**KOD KORZYŚCI**

ISBN 978-83-246-3372-2



Cena: 89,00 zł

**Informatyka w najlepszym wydaniu**