

Java

Kompendium programisty

Wydanie XII



Herbert Schildt



Mc
Graw
Hill

Tytuł oryginału: Java: The Complete Reference, Twelfth Edition

Tłumaczenie: Piotr Rajca, Mikołaj Szczepaniak, Rafał Jońca

ISBN: 978-83-8322-156-4

Original edition copyright © 2022 by McGraw Hill. All rights reserved.

Polish edition copyright © 2023 by Helion S.A. All rights reserved.

McGraw Hill, the McGraw Hill Publishing logo, The Complete Reference™, and related trade dress are trademarks or registered trademarks of McGraw Hill and/or its affiliates in the United States and other countries and may not be used without written permission. All other trademarks are the property of their respective owners. McGraw Hill is not associated with any product or vendor mentioned in this book.

Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. All other trademarks are the property of their respective owners, and McGraw Hill makes no claim of ownership by the mention of products that contain these marks.

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 wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: helion@helion.pl

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/javk12>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

O autorze	23
O redaktorze merytorycznym	24
Przedmowa	25

CZĘŚĆ I Język Java

1 Historia i ewolucja języka Java	31
Rodowód Javy	31
Narodziny nowoczesnego języka — C	31
Język C++ — następny krok	33
Podwaliny języka Java	33
Powstanie języka Java	33
Powiązanie z językiem C#	35
Jak Java wywarła wpływ na internet	35
Aplety Javy	35
Bezpieczeństwo	36
Przenośność	36
Magia języka Java — kod bajtowy	36
Wychodząc poza aplety	37
Szybszy harmonogram udostępniania	38
Serwlety — Java po stronie serwera	39
Hasła języka Java	39
Prostota	40
Obiektowość	40
Niezawodność	40
Wielowątkowość	41
Neutralność architektury	41
Interpretowalność i wysoka wydajność	41
Rozproszenie	41
Dynamika	41
Ewolucja Javy	41
Kultura innowacji	46

2	Podstawy języka Java	47
	Programowanie obiektowe	47
	Dwa paradygmaty	47
	Abstrakcja	48
	Trzy zasady programowania obiektowego	48
	Pierwszy przykładowy program	52
	Wpisanie kodu programu	52
	Kompilacja programów	53
	Blizsze spojrzenie na pierwszy przykładowy program	53
	Drugi prosty program	55
	Dwie instrukcje sterujące	56
	Instrukcja if	57
	Pętla for	58
	Bloki kodu	59
	Kwestie składniowe	60
	Znaki białe	60
	Identyfikatory	60
	Stałe	60
	Komentarze	61
	Separatory	61
	Słowa kluczowe języka Java	61
	Biblioteki klas Javy	62
3	Typy danych, zmienne i tablice	63
	Java to język ze ścisłą kontrolą typów	63
	Typy proste	63
	Typy całkowitoliczbowe	64
	Typ byte	64
	Typ short	65
	Typ int	65
	Typ long	65
	Typy zmiennoprzecinkowe	65
	Typ float	66
	Typ double	66
	Typ znakowy	66
	Typ logiczny	68
	Blizsze spojrzenie na stałe	68
	Stałe całkowitoliczbowe	68
	Stałe zmiennoprzecinkowe	69
	Stałe logiczne	70
	Stałe znakowe	70
	Stałe łańcuchowe	71
	Zmienne	71
	Deklaracja zmiennej	71
	Inicjalizacja dynamiczna	72
	Zasięg i czas życia zmiennych	72
	Konwersja typów i rzutowanie	74
	Automatyczna konwersja typów	74
	Rzutowanie niezgodnych typów	75
	Automatyczne rozszerzanie typów w wyrażeniach	76
	Zasady rozszerzania typu	76
	Tablice	77
	Tablice jednowymiarowe	77
	Tablice wielowymiarowe	79
	Alternatywna składnia deklaracji tablicy	82

Wnioskowanie typów zmiennych lokalnych	83
Ograniczenia var	84
Kilka słów o łańcuchach	85
4 Operatory	86
Operatory arytmetyczne	86
Podstawowe operatory arytmetyczne	87
Operator reszty z dzielenia	87
Operatory arytmetyczne z przypisaniem	88
Inkrementacja i dekrementacja	89
Operatory bitowe	90
Logiczne operatory bitowe	91
Przesunięcie w lewo	93
Przesunięcie w prawo	94
Przesunięcie w prawo bez znaku	95
Operatory bitowe z przypisaniem	96
Operatory relacji	97
Operatory logiczne	98
Operatory logiczne ze skracaniem	99
Operator przypisania	100
Operator ?	100
Kolejność wykonywania operatorów	101
Stosowanie nawiasów okrągłych	101
5 Instrukcje sterujące	103
Instrukcje wyboru	103
Instrukcja if	103
Tradycyjna instrukcja switch	106
Instrukcje iteracyjne	110
Pętla while	110
Pętla do-while	111
Pętla for	113
Wersja for-each pętli for	116
Wnioskowanie typów zmiennych lokalnych w pętlach for	120
Pętle zagnieżdżone	121
Instrukcje skoku	122
Instrukcja break	122
Instrukcja continue	125
Instrukcja return	126
6 Wprowadzenie do klas	128
Klasy	128
Ogólna postać klasy	128
Prosta klasa	129
Deklarowanie obiektów	131
Bliższe spojrzenie na operator new	132
Przypisywanie zmiennych referencyjnych do obiektów	132
Wprowadzenie do metod	133
Dodanie metody do klasy Box	133
Zwracanie wartości	135
Dodanie metody przyjmującej parametry	136
Konstruktor	138
Konstruktor sparametryzowany	139
Słowo kluczowe this	140
Ukrywanie zmiennych składowych	140

Mechanizm odzyskiwania pamięci	141
Klasa stosu	141
7 Dokładniejsze omówienie metod i klas	144
Przeciążanie metod	144
Przeciążanie konstruktorów	146
Obiekty jako parametry	148
Dokładniejsze omówienie przekazywania argumentów	150
Zwracanie obiektów	151
Rekurencja	152
Wprowadzenie do kontroli dostępu	154
Składowe statyczne	157
Słowo kluczowe final	158
Powtórka z tablic	159
Klasy zagnieżdżone i klasy wewnętrzne	160
Omówienie klasy String	162
Wykorzystanie argumentów wiersza poleceń	164
Zmienna liczba argumentów	165
Przeciążanie metod o zmiennej liczbie argumentów	167
Zmienna liczba argumentów i niejednoznaczności	168
Stosowanie wnioskowania typów zmiennych lokalnych z typami referencyjnymi	170
8 Dziedziczenie	172
Podstawy dziedziczenia	172
Dostęp do składowych a dziedziczenie	173
Bardziej praktyczny przykład	174
Zmienna klasy bazowej może zawierać referencję do obiektu klasy pochodnej	176
Słowo kluczowe super	177
Wykorzystanie słowa kluczowego super do wywołania konstruktora klasy bazowej	177
Drugie zastosowanie słowa kluczowego super	180
Tworzenie hierarchii wielopoziomowej	180
Kiedy są wykonywane konstruktory?	183
Przesłanianie metod	184
Dynamiczne przydzielanie metod	186
Dlaczego warto przesłaniać metody?	187
Zastosowanie przesłaniania metod	187
Klasy abstrakcyjne	189
Słowo kluczowe final i dziedziczenie	191
Słowo kluczowe final zapobiega przesłanianiu	191
Słowo kluczowe final zapobiega dziedziczeniu	192
Wnioskowanie typów zmiennych lokalnych a dziedziczenie	192
Klasa Object	193
9 Pakiety i interfejsy	195
Pakiety	195
Definiowanie pakietu	195
Znajdowanie pakietów i ścieżka CLASSPATH	196
Prosty przykład pakietu	197
Dostęp do pakietów i składowych	197
Przykład dostępu	198
Import pakietów	201
Interfejsy	202
Definiowanie interfejsu	203
Implementacja interfejsu	204
Interfejsy zagnieżdżone	206
Stosowanie interfejsów	207

Zmienne w interfejsach	209
Interfejsy można rozszerzać	211
Metody domyślne	211
Podstawy metod domyślnych	212
Bardziej praktyczny przykład	214
Problemy wielokrotnego dziedziczenia	214
Metody statyczne w interfejsach	215
Stosowanie metod prywatnych w interfejsach	216
Ostatnie uwagi dotyczące pakietów i interfejsów	217
10 Obsługa wyjątków	218
Podstawy obsługi wyjątków	218
Typy wyjątków	219
Nieprzechwycone wyjątki	219
Stosowanie instrukcji try i catch	220
Wyświetlenie opisu wyjątku	221
Wiele klauzul catch	222
Zagnieżdżone instrukcje try	223
Instrukcja throw	225
Klauzula throws	226
Słowo kluczowe finally	227
Wyjątki wbudowane w język Java	228
Tworzenie własnej klasy pochodnej wyjątków	229
Łańcuch wyjątków	231
Trzy dodatkowe cechy wyjątków	232
Wykorzystanie wyjątków	233
11 Programowanie wielowątkowe	234
Model wątków języka Java	235
Priorytety wątków	236
Synchronizacja	236
Przekazywanie komunikatów	236
Klasa Thread i interfejs Runnable	237
Wątek główny	237
Tworzenie wątku	238
Implementacja interfejsu Runnable	239
Rozszerzanie klasy Thread	240
Wybór odpowiedniego podejścia	241
Tworzenie wielu wątków	241
Stosowanie metod isAlive() i join()	243
Priorytety wątków	245
Synchronizacja	245
Synchronizacja metod	246
Instrukcja synchronized	247
Komunikacja międzywątkowa	249
Zakleszczenie	252
Zawieszanie, wznawianie i zatrzymywanie wątków	254
Uzyskiwanie stanu wątku	256
Stosowanie metody wytwórczej do tworzenia i uruchamiania wątku	257
Korzystanie z wielowątkowości	258
12 Wyliczenia, automatyczne opakowywanie typów prostych i adnotacje	259
Typy wyliczeniowe	259
Podstawy wyliczeń	259
Metody values() i valueOf()	261

Wyliczenia Javy jako typy klasowe	262
Wyliczenia dziedziczą po klasie Enum	264
Inny przykład wyliczenia	265
Opakowania typów	267
Klasa Character	267
Klasa Boolean	267
Opakowania typów numerycznych	268
Automatyczne opakowywanie typów prostych	269
Automatyczne opakowywanie i metody	270
Automatyczne opakowywanie i rozpakowywanie w wyrażeniach	270
Automatyczne opakowywanie typów znakowych i logicznych	272
Automatyczne opakowywanie pomaga zapobiegać błędom	272
Słowo ostrzeżenia	273
Adnotacje	273
Podstawy tworzenia adnotacji	273
Określanie strategii zachowywania adnotacji	274
Odczytywanie adnotacji w trakcie działania programu za pomocą refleksji	275
Interfejs AnnotatedElement	279
Wartości domyślne	279
Adnotacje znacznikowe	280
Adnotacje jednoelementowe	281
Wbudowane adnotacje	282
Adnotacje typów	284
Adnotacje powtarzalne	288
Ograniczenia	289
13 Wejście-wyjście, instrukcja try z zasobami i inne tematy	290
Podstawowa obsługa wejścia i wyjścia	290
Strumienie	291
Strumienie znakowe i bajtowe	291
Predefiniowane strumienie	293
Odczyt danych z konsoli	293
Odczyt znaków	294
Odczyt łańcuchów	295
Wyświetlanie informacji na konsoli	296
Klasa PrintWriter	297
Odczyt i zapis plików	297
Automatyczne zamykanie pliku	303
Modyfikatory transient i volatile	306
Prezentacja operatora instanceof	306
Modyfikator strictfp	308
Metody napisane w kodzie rdzennym	308
Stosowanie asercji	309
Opcje włączania i wyłączania asercji	311
Import statyczny	311
Wywoływanie przeciążonych konstruktorów za pomocą this()	313
Kilka słów o klasach wartościowych	315
14 Typy sparametryzowane	316
Czym są typy sparametryzowane?	316
Prosty przykład zastosowania typów sparametryzowanych	317
Typy sparametryzowane działają tylko dla typów referencyjnych	320
Typy sparametryzowane różnią się, jeśli mają inny argument typu	320
W jaki sposób typy sparametryzowane zwiększają bezpieczeństwo?	320
Klasa sparametryzowana z dwoma parametrami typu	322

Ogólna postać klasy sparametryzowanej	323
Typy ograniczone	323
Zastosowanie argumentów wieloznacznych	325
Ograniczony argument wieloznaczny	328
Tworzenie metody sparametryzowanej	332
Konstruktory sparametryzowane	333
Interfejsy sparametryzowane	334
Typy surowe i starszy kod	336
Hierarchia klas sparametryzowanych	338
Zastosowanie sparametryzowanej klasy bazowej	338
Podklasa sparametryzowana	340
Porównywanie typów w hierarchii klas sparametryzowanych w czasie wykonywania	341
Rzutowanie	342
Przesłanianie metod w klasach sparametryzowanych	343
Wnioskowanie typów a typy sparametryzowane	344
Wnioskowanie typów zmiennych lokalnych a typy sparametryzowane	345
Znoszenie	345
Metody mostu	345
Błędy niejednoznaczności	347
Pewne ograniczenia typów sparametryzowanych	348
Nie można tworzyć obiektu parametru typu	348
Ograniczenia dla składowych statycznych	348
Ograniczenia tablic typów sparametryzowanych	348
Ograniczenia wyjątków typów sparametryzowanych	349
15 Wyrażenia lambda	350
Wprowadzenie do wyrażań lambda	350
Podstawowe informacje o wyrażeniach lambda	351
Interfejsy funkcyjne	351
Kilka przykładów wyrażań lambda	352
Blokowe wyrażenia lambda	355
Sparametryzowane interfejsy funkcyjne	357
Przekazywanie wyrażań lambda jako argumentów	358
Wyrażenia lambda i wyjątki	361
Wyrażenia lambda i przechwytywanie zmiennych	362
Referencje do metod	363
Referencje do metod statycznych	363
Referencje do metod instancyjnych	364
Referencje do metod a typy sparametryzowane	367
Referencje do konstruktorów	369
Predefiniowane interfejsy funkcyjne	373
16 Moduły	375
Podstawowe informacje o modułach	375
Przykład prostego modułu	376
Kompilowanie i uruchamianie przykładowej aplikacji	379
Dokładniejsze informacje o instrukcjach requires i exports	380
java.base i moduły platformy	381
Stary kod i moduł nienazwany	382
Eksportowanie do konkretnego modułu	383
Wymagania przechodnie	384
Stosowanie usług	388
Podstawowe informacje o usługach i dostawcach usług	388
Słowa kluczowe związane z usługami	388
Przykład stosowania usług i modułów	389

Grafy modułów	395
Trzy wyspecjalizowane cechy modułów	395
Moduły otwarte	395
Instrukcja opens	396
Instrukcja requires static	396
Wprowadzenie do jlink i plików JAR modułów	396
Dołączanie plików dostarczonych jako struktura katalogów	396
Konsolidacja modularnych plików JAR	397
Pliki JMOD	398
Kilka słów o warstwach i modułach automatycznych	398
Końcowe uwagi dotyczące modułów	398

17 Wyrażenia switch, rekordy oraz inne najnowsze możliwości języka 399

Rozszerzenia switch	400
Listy stałych	401
Wprowadzanie wyrażenia switch i instrukcji yield	402
Wprowadzenie strzałek do klauzul case	403
Dokładniejsza prezentacja zapisu ze strzałką	405
Inny przykład wyrażenia switch	408
Bloki tekstu	408
Podstawowe informacje o blokach tekstu	408
Traktowanie początkowych białych znaków	409
Użycie cudzysłowów w blokach tekstu	411
Sekwencje specjalne w blokach tekstu	411
Rekordy	412
Podstawowe informacje o rekordach	412
Tworzenie konstruktorów rekordów	414
Inny przykład konstruktora rekordu	418
Tworzenie metod pobierających w rekordach	419
Dopasowywanie wzorców z użyciem operatora instanceof	420
Zmienne wzorców i logiczne wyrażenia AND	421
Dopasowywanie wzorców w innych wyrażeniach	422
Klasy i interfejsy „zapięczętowane”	423
Klasy zapięczętowane	423
Interfejsy zapięczętowane	425
Kierunki rozwoju	426

CZĘŚĆ II

Biblioteka języka Java

18 Obsługa łańcuchów 429

Konstruktory klasy String	429
Długość łańcucha	431
Specjalne operacje na łańcuchach	431
Literały tekstowe	431
Konkatenacja łańcuchów	432
Konkatenacja łańcuchów z innymi typami danych	432
Konwersja łańcuchów i metoda toString()	433
Wyodrębnianie znaków	434
Metoda charAt()	434
Metoda getChars()	434
Metoda getBytes()	434
Metoda toCharArray()	435

Porównywanie łańcuchów	435
Metody equals() i equalsIgnoreCase()	435
Metoda regionMatches()	436
Metody startsWith() i endsWith()	436
Metoda equals() kontra operator ==	436
Metoda compareTo()	437
Przeszukiwanie łańcuchów	438
Modyfikowanie łańcucha	439
Metoda substring()	439
Metoda concat()	440
Metoda replace()	440
Metody trim() i strip()	441
Konwersja danych za pomocą metody valueOf()	442
Zmiana wielkości liter w łańcuchu	442
Łączenie łańcuchów	443
Dodatkowe metody klasy String	443
Klasa StringBuffer	445
Konstruktory klasy StringBuffer	445
Metody length() i capacity()	445
Metoda ensureCapacity()	446
Metoda setLength()	446
Metody charAt() i setCharAt()	446
Metoda getChars()	447
Metoda append()	447
Metoda insert()	447
Metoda reverse()	448
Metody delete() i deleteCharAt()	448
Metoda replace()	449
Metoda substring()	449
Dodatkowe metody klasy StringBuffer	449
Klasa StringBuilder	450
19 Pakiet java.lang	451
Opakowania typów prostych	451
Klasa Number	452
Klasy Double i Float	452
Metody isInfinite() i isNaN()	455
Klasy Byte, Short, Integer i Long	456
Klasa Character	464
Dodatki wprowadzone w celu obsługi punktów kodowych Unicode	466
Klasa Boolean	467
Klasa Void	468
Klasa Process	468
Klasa Runtime	469
Wykonywanie innych programów	470
Runtime.Version	471
Klasa ProcessBuilder	473
Klasa System	475
Wykorzystanie metody currentTimeMillis()	
do obliczania czasu wykonywania programu	476
Użycie metody arraycopy()	477
Właściwości środowiska	477
Interfejs System.Logger i klasa System.LoggerFinder	478
Klasa Object	478

Wykorzystanie metody clone() i interfejsu Cloneable	478
Klasa Class	480
Klasa ClassLoader	483
Klasa Math	483
Funkcje trygonometryczne	483
Funkcje wykładnicze	484
Funkcje zaokrągleń	484
Inne metody klasy Math	485
Klasa StrictMath	487
Klasa Compiler	487
Klasy Thread i ThreadGroup oraz interfejs Runnable	487
Interfejs Runnable	488
Klasa Thread	488
Klasa ThreadGroup	490
Klasy ThreadLocal i InheritableThreadLocal	493
Klasa Package	493
Klasa Module	494
Klasa ModuleLayer	495
Klasa RuntimePermission	495
Klasa Throwable	495
Klasa SecurityManager	495
Klasa StackTraceElement	495
Klasa StackWalker i interfejs StackWalker.StackFrame	496
Klasa Enum	496
Klasa Record	497
Klasa ClassValue	497
Interfejs CharSequence	497
Interfejs Comparable	498
Interfejs Appendable	498
Interfejs Iterable	499
Interfejs Readable	499
Interfejs AutoCloseable	499
Interfejs Thread.UncaughtExceptionHandler	500
Podpakiety pakietu java.lang	500
Podpakiet java.lang.annotation	500
Podpakiet java.lang.constant	500
Podpakiet java.lang.instrument	500
Podpakiet java.lang.invoke	500
Podpakiet java.lang.management	500
Podpakiet java.lang.module	501
Podpakiet java.lang.ref	501
Podpakiet java.lang.reflect	501

20 Pakiet java.util, część 1. — kolekcje 502

Wprowadzenie do kolekcji	503
Interfejsy kolekcji	504
Interfejs Collection	505
Interfejs List	507
Interfejs Set	508
Interfejs SortedSet	509
Interfejs NavigableSet	509
Interfejs Queue	510
Interfejs Deque	511

Klasy kolekcji	512
Klasa ArrayList	513
Klasa LinkedList	516
Klasa HashSet	517
Klasa LinkedHashSet	518
Klasa TreeSet	518
Klasa PriorityQueue	519
Klasa ArrayDeque	520
Klasa EnumSet	521
Dostęp do kolekcji za pomocą iteratora	521
Korzystanie z iteratora Iterator	523
Pętla typu for-each jako alternatywa dla iteratora	524
Spliterator	525
Przechowywanie w kolekcjach własnych klas	527
Interfejs RandomAccess	529
Korzystanie z map	529
Interfejsy map	529
Klasy map	534
Komparatory	538
Stosowanie komparatora	540
Algorytmy kolekcji	545
Klasa Arrays	550
Starsze klasy i interfejsy	554
Interfejs Enumeration	554
Klasa Vector	555
Klasa Stack	558
Klasa Dictionary	559
Klasa Hashtable	560
Klasa Properties	563
Wykorzystanie metod store() i load()	566
Ostatnie uwagi na temat kolekcji	567
21 Pakiet java.util, część 2. — pozostałe klasy użytkowe	568
Klasa StringTokenizer	568
Klasa BitSet	570
Klasy Optional, OptionalDouble, OptionalInt oraz OptionalLong	572
Klasa Date	575
Klasa Calendar	576
Klasa GregorianCalendar	579
Klasa TimeZone	580
Klasa SimpleTimeZone	581
Klasa Locale	582
Klasa Random	583
Klasy Timer i TimerTask	585
Klasa Currency	587
Klasa Formatter	588
Konstruktory klasy Formatter	589
Metody klasy Formatter	589
Podstawy formatowania	590
Formatowanie łańcuchów i znaków	592
Formatowanie liczb	592
Formatowanie daty i godziny	593
Specyfikatory %n i %%	595
Określanie minimalnej szerokości pola	595

Określanie precyzji	596
Używanie znaczników (flag) formatów	597
Wyrównywanie danych wyjściowych	597
Znaczniki spacji, plusa, zera i nawiasów	598
Znacznik przecinka	599
Znacznik #	599
Opcja wielkich liter	599
Stosowanie indeksu argumentu	600
Zamykanie obiektu klasy Formatter	601
Metoda printf() w Javie	602
Klasa Scanner	602
Konstruktory klasy Scanner	602
Podstawy skanowania	603
Kilka przykładów użycia klasy Scanner	606
Ustawianie separatorów	609
Pozostałe elementy klasy Scanner	610
Klasy ResourceBundle, ListResourceBundle i PropertyResourceBundle	611
Dodatkowe klasy i interfejsy użytkowe	615
Podpakiety pakietu java.util	616
java.util.concurrent, java.util.concurrent.atomic oraz java.util.concurrent.locks	616
java.util.function	617
java.util.jar	619
java.util.logging	619
java.util.prefs	619
java.util.random	619
java.util.regex	619
java.util.spi	620
java.util.stream	620
java.util.zip	620
22 Operacje wejścia-wyjścia: analiza pakietu java.io	621
Klasy i interfejsy obsługujące operacje wejścia-wyjścia	622
Klasa File	622
Katalogi	625
Stosowanie interfejsu FilenameFilter	626
Alternatywna metoda listFiles()	627
Tworzenie katalogów	627
Interfejsy AutoCloseable, Closeable i Flushable	627
Wyjątki operacji wejścia-wyjścia	628
Dwa sposoby zamykania strumieni	628
Klasy strumieni	629
Strumienie bajtów	630
Klasa InputStream	630
Klasa OutputStream	631
Klasa FileInputStream	632
Klasa FileOutputStream	633
Klasa ByteArrayInputStream	635
Klasa ByteArrayOutputStream	637
Filtrowane strumienie bajtów	638
Buforowane strumienie bajtów	638
Klasa SequenceInputStream	641
Klasa PrintStream	643
Klasy DataOutputStream i DataInputStream	645
Klasa RandomAccessFile	646

Strumienie znaków	647
Klasa Reader	647
Klasa Writer	647
Klasa FileReader	649
Klasa FileWriter	649
Klasa CharArrayReader	650
Klasa CharArrayWriter	651
Klasa BufferedReader	652
Klasa BufferedWriter	654
Klasa PushbackReader	654
Klasa PrintWriter	655
Klasa Console	656
Serializacja	658
Interfejs Serializable	658
Interfejs Externalizable	658
Interfejs ObjectOutputStream	659
Klasa ObjectOutputStream	659
Interfejs ObjectInputStream	660
Klasa ObjectInputStream	661
Przykład serializacji	662
Korzyści wynikające ze stosowania strumieni	665
23 System NIO	666
Klasy systemu NIO	666
Podstawy systemu NIO	667
Bufory	667
Kanały	669
Zestawy znaków i selektory	670
Udoskonalenia dodane w systemie NIO.2	671
Interfejs Path	671
Klasa Files	672
Klasa Paths	674
Interfejsy atrybutów plików	675
Klasy FileSystem, FileSystems i FileStore	676
Stosowanie systemu NIO	677
Stosowanie systemu NIO dla operacji wejścia-wyjścia na kanałach	677
Stosowanie systemu NIO dla operacji wejścia-wyjścia na strumieniach	685
Stosowanie systemu NIO dla operacji na ścieżkach i systemie plików	687
24 Obsługa sieci	695
Podstawy działania sieci	695
Klasy i interfejsy pakietu java.net obsługujące komunikację sieciową	696
Klasa InetAddress	697
Metody wytwórcze	697
Metody klasy	698
Klasy InetAddress oraz Inet6Address	699
Gniazda klientów TCP/IP	699
URL	702
Klasa URLConnection	703
Klasa HttpURLConnection	705
Klasa URI	707
Pliki cookie	707
Gniazda serwerów TCP/IP	707
Datagramy	708

Klasa DatagramSocket	708
Klasa DatagramPacket	709
Przykład użycia datagramów	710
Prezentacja pakietu java.net.http	711
Trzy kluczowe elementy	712
Prosty przykład użycia API klienta HTTP	714
Czego jeszcze warto dowiedzieć się o pakiecie java.net.http?	716
25 Obsługa zdarzeń	717
Dwa mechanizmy obsługi zdarzeń	717
Model obsługi zdarzeń oparty na ich delegowaniu	718
Zdarzenia	718
Źródła zdarzeń	718
Obiekty nasłuchujące zdarzeń	719
Klasy zdarzeń	719
Klasa ActionEvent	721
Klasa AdjustmentEvent	721
Klasa ComponentEvent	722
Klasa ContainerEvent	722
Klasa FocusEvent	723
Klasa InputEvent	724
Klasa ItemEvent	724
Klasa KeyEvent	725
Klasa MouseEvent	726
Klasa MouseWheelEvent	727
Klasa TextEvent	728
Klasa WindowEvent	728
Źródła zdarzeń	729
Interfejsy nasłuchujące zdarzeń	730
Interfejs ActionListener	731
Interfejs AdjustmentListener	731
Interfejs ComponentListener	731
Interfejs ContainerListener	731
Interfejs FocusListener	731
Interfejs ItemListener	731
Interfejs KeyListener	731
Interfejs MouseListener	732
Interfejs MouseMotionListener	732
Interfejs MouseWheelListener	732
Interfejs TextListener	732
Interfejs WindowFocusListener	732
Interfejs WindowListener	732
Stosowanie modelu delegowania zdarzeń	733
Kluczowe zagadnienia tworzenia aplikacji graficznych z użyciem AWT	733
Obsługa zdarzeń generowanych przez mysz	734
Obsługa zdarzeń generowanych przez klawiaturę	737
Klasy adapterów	740
Klasy wewnętrzne	742
Anonimowa klasa wewnętrzna	744
26 Wprowadzenie do AWT: praca z oknami, grafiką i tekstem	746
Klasy AWT	747
Podstawy okien	749
Klasa Component	749
Klasa Container	749

Klasa Panel	749
Klasa Window	750
Klasa Frame	750
Klasa Canvas	750
Praca z oknami typu Frame	750
Ustawianie wymiarów okna	750
Ukrywanie i wyświetlanie okna	751
Ustawianie tytułu okna	751
Zamykanie okna typu Frame	751
Metoda paint()	751
Wyświetlanie łańcuchów znaków	751
Określanie koloru tekstu i tła	752
Żądanie ponownego wyświetlenia zawartości okna	752
Tworzenie aplikacji korzystających z klasy Frame	753
Wprowadzenie do stosowania grafiki	754
Rysowanie odcinków	754
Rysowanie prostokątów	754
Rysowanie elips, kół i okręgów	755
Rysowanie łuków	755
Rysowanie wielokątów	755
Prezentacja metod rysujących	756
Dostosowywanie rozmiarów obiektów graficznych	757
Praca z klasą Color	758
Metody klasy Color	759
Ustawianie bieżącego koloru kontekstu graficznego	760
Program demonstrujący zastosowanie klasy Color	760
Ustawianie trybu rysowania	761
Praca z czcionkami	762
Określanie dostępnych czcionek	763
Tworzenie i wybieranie czcionek	765
Uzyskiwanie informacji o czcionkach	767
Zarządzanie tekstowymi danymi wyjściowymi z wykorzystaniem klasy FontMetrics	768
27 Stosowanie kontroltek AWT, menedżerów układu graficznego oraz menu	771
Podstawy kontroltek AWT	772
Dodawanie i usuwanie kontroltek	772
Odpowiadanie na zdarzenia kontroltek	772
Wyjątek HeadlessException	773
Etykiety	773
Stosowanie przycisków	774
Obsługa zdarzeń przycisków	775
Stosowanie pól wyboru	778
Obsługa zdarzeń pól wyboru	779
Klasa CheckboxGroup	780
Kontrolki list rozwijanych	782
Obsługa zdarzeń list rozwijanych	783
Stosowanie list	784
Obsługa zdarzeń generowanych przez listy	785
Zarządzanie paskami przewijania	787
Obsługa zdarzeń generowanych przez paski przewijania	788
Stosowanie kontroltek typu TextField	790
Obsługa zdarzeń generowanych przez kontrolkę TextField	791
Stosowanie kontroltek typu TextArea	792

Wprowadzenie do menedżerów układu graficznego komponentów	794
FlowLayout	795
BorderLayout	796
Stosowanie obramowań	797
GridLayout	799
Klasa CardLayout	800
Klasa GridBagLayout	803
Menu i paski menu	807
Okna dialogowe	812
Przesłanie metody paint()	815
28 Obrazy	817
Formaty plików	817
Podstawy przetwarzania obrazów: tworzenie, wczytywanie i wyświetlanie	818
Tworzenie obiektu obrazu	818
Ładowanie obrazu	818
Wyświetlanie obrazu	819
Podwójne buforowanie	820
Interfejs ImageProducer	822
Klasa MemoryImageSource	822
Interfejs ImageConsumer	825
Klasa PixelGrabber	825
Klasa ImageFilter	827
Klasa CropImageFilter	828
Klasa RGBImageFilter	830
Dodatkowe klasy obsługujące obrazy	840
29 Narzędzia współbieżności	841
Pakiety interfejsu Concurrent API	842
Pakiet java.util.concurrent	842
Pakiet java.util.concurrent.atomic	843
Pakiet java.util.concurrent.locks	843
Korzystanie z obiektów służących do synchronizacji	843
Klasa Semaphore	843
Klasa CountDownLatch	848
CyclicBarrier	850
Klasa Exchanger	852
Klasa Phaser	853
Korzystanie z egzekutorów	860
Przykład prostego egzekutora	860
Korzystanie z interfejsów Callable i Future	862
Typ wyliczeniowy TimeUnit	864
Kolekcje współbieżne	865
Blokady	865
Operacje atomowe	868
Programowanie równoległe przy użyciu frameworku Fork/Join	869
Najważniejsze klasy frameworku Fork/Join	870
Strategia dziel i zwyciężaj	873
Prosty przykład użycia frameworku Fork/Join	873
Znaczenie poziomu równoległości	875
Przykład użycia klasy RecursiveTask<V>	878
Asynchroniczne wykonywanie zadań	880
Anulowanie zadania	880
Określanie statusu wykonania zadania	881

Ponowne uruchamianie zadania	881
Pozostałe zagadnienia	881
Wskazówki dotyczące stosowania frameworku Fork/Join	883
Pakiet Concurrency Utilities a tradycyjne metody języka Java	883
30 API strumieni	884
Podstawowe informacje o strumieniach	884
Interfejsy strumieni	885
Jak można uzyskać strumień?	887
Prosty przykład stosowania strumieni	888
Operacje redukcji	891
Stosowanie strumieni równoległych	893
Odwzorowywanie	895
Tworzenie kolekcji	899
Iteratory i strumienie	902
Stosowanie typu Iterator i strumieni	902
Stosowanie spliteratorów	903
Inne możliwości API strumieni	906
31 Wyrażenia regularne i inne pakiety	907
Przetwarzanie wyrażeń regularnych	907
Klasa Pattern	908
Klasa Matcher	908
Składnia wyrażeń regularnych	909
Przykład dopasowywania do wzorca	909
Dwie opcje dopasowywania do wzorca	914
Przegląd wyrażeń regularnych	914
Refleksja	914
Zdalne wywoływanie metod (RMI)	918
Prosta aplikacja typu klient-serwer wykorzystująca RMI	918
Formatowanie dat i czasu przy użyciu pakietu java.text	921
Klasa DateFormat	921
Klasa SimpleDateFormat	922
Interfejs API dat i czasu — java.time	924
Podstawowe klasy do obsługi dat i czasu	924
Formatowanie dat i godzin	926
Analiza łańcuchów zawierających daty i godziny	928
Inne możliwości pakietu java.time	928

CZĘŚĆ III

Wprowadzenie do programowania GUI przy użyciu pakietu Swing

32 Wprowadzenie do pakietu Swing	933
Geneza powstania biblioteki Swing	933
Bibliotekę Swing zbudowano na bazie zestawu narzędzi AWT	934
Podstawowe cechy biblioteki Swing	934
Komponenty biblioteki Swing są lekkie	934
Biblioteka Swing obsługuje dołączany wygląd i sposób obsługi	934
Podobieństwo do architektury MVC	935
Komponenty i kontenery	936
Komponenty	936
Kontenery	937
Panele kontenerów najwyższego poziomu	937
Pakiety biblioteki Swing	938

Prosta aplikacja na bazie biblioteki Swing	938
Obsługa zdarzeń	941
Rysowanie w bibliotece Swing	944
Podstawy rysowania	944
Wyznaczanie obszaru rysowania	945
Przykład rysowania	946
33 Przewodnik po pakiecie Swing	949
Klasy JLabel i ImageIcon	949
Klasa JTextField	951
Przyciski biblioteki Swing	953
Klasa JButton	953
Klasa JToggleButton	955
Pola wyboru	957
Przyciski opcji	958
Klasa JTabbedPane	960
Klasa JScrollPane	963
Klasa JList	964
Klasa JComboBox	967
Drzewa	969
Klasa JTable	972
34 Wprowadzenie do systemu menu pakietu Swing	975
Podstawy systemu menu	975
Przegląd klas JMenuBar, JMenu oraz JMenuItem	977
Klasa JMenuBar	977
Klasa JMenu	977
Klasa JMenuItem	978
Tworzenie menu głównego	979
Dodawanie mnemoników i kombinacji klawiszy do opcji menu	983
Dodawanie obrazów i etykiet ekranowych do menu	984
Stosowanie klas JRadioButtonMenuItem i JCheckBoxMenuItem	986
Tworzenie menu podręcznych	988
Tworzenie paska narzędzi	991
Stosowanie akcji	993
Finalna postać programu MenuDemo	998
Dalsze poznawanie pakietu Swing	1003

CZĘŚĆ IV

Stosowanie Javy w praktyce

35 Java Beans	1007
Czym jest komponent typu Java Bean?	1007
Zalety komponentów Java Beans	1008
Introspekcja	1008
Wzorce właściwości	1008
Wzorce projektowe dla zdarzeń	1009
Metody i wzorce projektowe	1010
Korzystanie z interfejsu BeanInfo	1010
Właściwości ograniczone	1010
Trwałość	1011
Interfejs Customizer	1011
Interfejs Java Beans API	1011
Klasa Introspector	1013
Klasa PropertyDescriptor	1013

Klasa EventSetDescriptor	1013
Klasa MethodDescriptor	1013
Przykład komponentu Java Bean	1013
36 Serwlety	1017
Podstawy	1017
Cykl życia serwletu	1018
Sposoby tworzenia serwletów	1018
Korzystanie z serwera Tomcat	1019
Przykład prostego serwletu	1020
Tworzenie i kompilacja kodu źródłowego serwletu	1020
Uruchamianie serwera Tomcat	1021
Uruchamianie przeglądarki i generowanie żądania	1021
Interfejs Servlet API	1021
Pakiet javax.servlet	1021
Interfejs Servlet	1022
Interfejs ServletConfig	1022
Interfejs ServletContext	1023
Interfejs ServletRequest	1023
Interfejs ServletResponse	1024
Klasa GenericServlet	1024
Klasa ServletInputStream	1024
Klasa ServletOutputStream	1025
Klasy wyjątków związanych z serwletami	1025
Odczytywanie parametrów serwletu	1025
Pakiet jakarta.servlet.http	1026
Interfejs HttpServletRequest	1027
Interfejs HttpServletResponse	1027
Interfejs HttpSession	1028
Klasa Cookie	1029
Klasa HttpServlet	1030
Obsługa żądań i odpowiedzi HTTP	1031
Obsługa żądań GET protokołu HTTP	1031
Obsługa żądań POST protokołu HTTP	1032
Korzystanie ze znaczników kontekstu użytkownika	1033
Śledzenie sesji	1035

DODATKI

A Komentarze dokumentujące	1039
Znaczniki narzędzia javadoc	1039
Znacznik @author	1040
Znacznik {@code}	1040
Znacznik @deprecated	1041
Znacznik {@docRoot}	1041
Znacznik @exception	1041
Znacznik @hidden	1041
Znacznik {@index}	1041
Znacznik {@inheritDoc}	1041
Znacznik {@link}	1041
Znacznik {@linkplain}	1042
Znacznik {@literal}	1042
Znacznik @param	1042
Znacznik @provides	1042

Znacznik @return	1042
Znacznik @see	1042
Znacznik @serial	1043
Znacznik @serialData	1043
Znacznik @serialField	1043
Znacznik @since	1043
Znacznik {@summary}	1043
Znacznik {@systemProperty}	1043
Znacznik @throws	1044
Znacznik @uses	1044
Znacznik {@value}	1044
Znacznik @version	1044
Ogólna postać komentarzy dokumentacyjnych	1044
Wynik działania narzędzia javadoc	1044
Przykład korzystający z komentarzy dokumentacyjnych	1045
B Wprowadzenie do JShell	1046
Podstawy JShell	1046
Wyświetlanie, edytowanie i ponowne wykonywanie kodu	1048
Dodanie metody	1049
Utworzenie klasy	1050
Stosowanie interfejsu	1050
Przetwarzanie wyrażeń i wbudowane zmienne	1051
Importowanie pakietów	1052
Wyjątki	1052
Inne polecenia JShell	1053
Dalsze poznawanie możliwości JShell	1054
C Kompilowanie i uruchamianie prostych programów w jednym kroku	1055
 Skorowidz	1057

ROZDZIAŁ

7

Dokładniejsze omówienie metod i klas

Niniejszy rozdział kontynuuje omówienie metod i klas, zapoczątkowane w poprzednim rozdziale. Porusza kilka interesujących tematów związanych z metodami, między innymi ich przeciążanie, przekazywanie parametrów i rekurencję. W dalszej części wrócimy do klas, a dokładnie do zarządzania dostępem, zastosowania słowa kluczowego `static` i jednej z podstawowych klas Javy — `String`.

Przeciążanie metod

W języku Java można zdefiniować w jednej klasie dwie lub więcej metod o takiej samej nazwie, o ile posiadają one inne deklaracje parametrów. W takim przypadku mówi się o **przeciążaniu metody**. Przeciążanie metod to jeden ze sposobów obsługi polimorfizmu w języku Java. Jeśli nigdy wcześniej nie używało się języka programowania dopuszczającego przeciążanie, koncepcja może się wydawać dziwaczna. W praktyce jednak nie jest szczególnie trudna i co ważniejsze, znacząco ułatwia użytkownikom korzystanie z metod.

W momencie wywołania przeciążonej metody Java potrzebuje typu i (lub) liczby argumentów do określenia, którą tak naprawdę wersję przeciążonej metody należy wykonać. Choć przeciążone metody mogą zwracać wartości różnych typów, znaczenie ma jedynie liczba i typ argumentów. Z tego powodu przeciążane metody muszą różnić się parametrami, a nie tylko typem zwracanej wartości. Java w momencie napotkania wywołania przeciążonej metody szuka wersji najlepiej pasującej do przekazywanych argumentów i wykonuje jej kod.

Następujący przykład obrazuje przeciążanie metod.

```
// R07\Overload.java
```

```
// Przykład przeciążania metod
```

```
class OverloadDemo {  
    void test() {  
        System.out.println("Brak parametrów");  
    }  
  
    // Przeciążona metoda test z jednym parametrem typu int  
    void test(int a) {  
        System.out.println("a: " + a);  
    }  
  
    // Przeciążona metoda test z dwoma parametrami  
    void test(int a, int b) {  
        System.out.println("a i b: " + a + " " + b);  
    }  
}
```

```
// Przeciążona metoda test z jednym parametrem typu double
double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
}

class Overload {
    public static void main(String[] args) {
        OverloadDemo ob = new OverloadDemo();
        double result;

        // Wywołanie wszystkich wersji metody test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Wynik wykonania ob.test(123.25): " + result);
    }
}
```

Program generuje następujące wyniki.

```
Brak parametrów
a: 10
a i b: 10 20
double a: 123.25
Wynik wykonania ob.test(123.25): 15190.5625
```

Metoda `test()` jest przeciążona cztery razy. Pierwsza wersja nie przyjmuje żadnych parametrów, druga przyjmuje jeden parametr typu `int`, trzecia przyjmuje dwa parametry, a czwarta przyjmuje jeden parametr typu `double`. Dodatkowo ostatnia wersja zwraca wartość typu `double`, ale nie wpływa to w żaden sposób na wybór odpowiedniej metody.

W momencie wywołania przeciążonej metody Java szuka dopasowania przekazanych argumentów z przyjmowanymi parametrami. Nie zawsze dopasowanie musi być dokładne. Pod uwagę brane są również typy zgodne. Rozważmy następujący program.

```
// R07\OverLoad2.java
// Przy wyborze przeciążonej metody Java stosuje również automatyczną konwersję typów
class OverloadDemo {
    void test() {
        System.out.println("Brak parametrów");
    }

    // Przeciążona wersja dla dwóch parametrów
    void test(int a, int b) {
        System.out.println("a i b: " + a + " " + b);
    }

    // Przeciążona wersja dla jednego parametru typu double
    void test(double a) {
        System.out.println("Wewnątrz test(double) a: " + a);
    }
}

class Overload2 {
    public static void main(String[] args) {
        OverloadDemo ob = new OverloadDemo();
        int i = 88;

        ob.test();
        ob.test(10, 20);

        ob.test(i); // Spowoduje wywołanie test(double)
        ob.test(123.2); // Spowoduje wywołanie test(double)
    }
}
```

Program po uruchomieniu generuje następujące wyniki.

```
Brak parametrów
a i b: 10 20
Wewnątrz test(double) a: 88.0
Wewnątrz test(double) a: 123.2
```

Przedstawiona wersja klasy `OverloadDemo` nie definiuje metody `test(int)`. Z tego powodu wywołanie metody `test()` z argumentem typu `int` nie powoduje znalezienia idealnie pasującej wersji. Ponieważ jednak Java dokonuje automatycznej konwersji typu `int` na `double`, zostanie znaleziona metoda `test(double)`. Zostanie jej przekazana wersja argumentu po konwersji do typu `double`. Oczywiście gdyby istniała wersja `test(int)`, to ona zostałaby wykonana dla argumentu typu `int`. Java stosuje automatyczną konwersję tylko wtedy, gdy nie istnieje idealne dopasowanie.

Przeciążanie metod to element polimorfizmu, gdyż to jedyne rozwiązanie zapewniające spełnienie w Javie paradygmatu „jeden interfejs, wiele metod”. Warto zauważyć następującą kwestię: w języku programowania nieobsługującym przeciążania metod trzeba by nadać każdej metodzie unikatową nazwę. Sytuacja, w której chce się zaimplementować tę samą metodę dla wielu różnych danych, pojawia się stosunkowo często. Rozważmy sytuację z metodą zwracającą wartość bezwzględną. W językach, które nie obsługują przeciążania, poszczególne wersje muszą mieć odrobinę inne nazwy (w języku C funkcja `abs()` dotyczy liczb całkowitych, `labs()` liczb `long`, natomiast `fabs()` liczb zmiennoprzecinkowych). Jak łatwo zauważyć, nie jest to rozwiązanie wygodne, a przy większych projektach staje się znaczącym problemem — choć wszystkie wersje funkcji wykonują to samo zadanie, trzeba pamiętać trzy nazwy. W Javie przedstawiony problem nie występuje — istnieje tylko jedna nazwa dla metody zwracającej wartość bezwzględną. Jest to metoda `abs()` z klasy `Math`. Klasa zawiera wiele przeciążonych wersji tej metody obsługujących wszystkie podstawowe typy numeryczne. Java identyfikuje właściwą wersję metody `abs()` na podstawie typu argumentu.

Głównym zadaniem przeciążania jest zebranie wszystkich powiązanych ze sobą metod pod jedną nazwą. Innymi słowy, `abs` oznacza ogólną **akcję wykonywaną** dla liczb. To na kompilator spada odpowiedzialność co do wyboru **odpowiedniej** wersji metody dla danych argumentów. My, programiści, musimy pamiętać jedynie ogólną operację, którą chcemy wykonać. Dzięki polimorfizmowi kilka nazw redukuje się do jednej. Choć przedstawiony przykład jest stosunkowo prosty, pozwala zauważyć, iż przeciążanie pomaga lepiej opanować zagadnienia o znacznej złożoności.

Każda z przeciążonych wersji metody może wykonywać dowolne zadania. Nie istnieje zasada mówiąca, iż przeciążone metody muszą działać podobnie. W praktyce jednak wzajemne powiązanie wersji występuje bardzo często, gdyż właśnie tego oczekuje użytkownik. Innymi słowy, nie ma wielkiego sensu, by metoda o nazwie `sqr()` zwracała całkowitą podniesioną do kwadratu i **pierwiastek kwadratowy** dla liczb zmiennoprzecinkowych, gdyż obie operacje są sobie przeciwne. Takie zastosowanie przeciążenia przeczyłoby jego podstawowej idei. Z przedstawionych powodów przeciążania metod używa się jedynie dla powiązanych ze sobą operacji.

Przeciążanie konstruktorów

Przeciążanie nie dotyczy tylko metod — może być stosowane również dla konstruktorów. W zasadzie w większości programów Javy przeciążone konstruktory to raczej norma niż wyjątek. Aby lepiej zrozumieć, dlaczego przeciążanie konstruktorów jest tak popularne, przyjrzyjmy się jeszcze raz przykładowi z pudełkami z poprzedniego programu. Oto wcześniej stosowana wersja klasy `Box`.

```
// R07\Box.java

class Box {
    double width;
    double height;
    double depth;

    // Konstruktor klasy Box
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
}
```

```
// Oblicz i zwróć objętość
double volume() {
    return width * height * depth;
}
}
```

Konstruktor `Box()` wymaga przekazania trzech parametrów. Oznacza to, że przy tworzeniu każdego z obiektów klasy `Box` trzeba podać początkowe wymiary pudełka. Dla powyższej klasy następujący wiersz nie jest poprawny.

```
Box ob = new Box();
```

Ponieważ konstruktor wymaga podania trzech argumentów, wywołanie go bez argumentów jest błędem. Pojawia się następujący problem: w jaki sposób utworzyć pudełko, gdy nie interesują nas początkowe wymiary lub ich nie znamy? A co, jeśli chcemy utworzyć sześcian, więc wystarczyłoby podanie tylko jednej wartości dla wszystkich trzech wymiarów? W aktualnej wersji klasy `Box` oba te podejścia nie są dostępne.

Na szczęście łatwo rozwiązać przedstawione problemy: wystarczy przeciążyć konstruktor klasy `Box`, aby spełniał wymienione wcześniej warunki. Poniżej znajduje się ulepszona wersja klasy `Box` zawierająca trzy wersje konstruktora.

```
// R07\OverloadCons.java
/* Klasa Box zawiera trzy konstruktory, które
   na różne sposoby inicjalizują wymiary pudełka
*/
class Box {
    double width;
    double height;
    double depth;

    // Konstruktor używany, gdy podano wszystkie wymiary
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // Konstruktor używany przy braku wymiarów
    Box() {
        width = -1; // Wartość -1
        height = -1; // wskazuje
        depth = -1; // niezainicjalizowane pudełko
    }

    // Konstruktor używany do tworzenia sześcianów
    Box(double len) {
        width = height = depth = len;
    }

    // Oblicz i zwróć objętość
    double volume() {
        return width * height * depth;
    }
}

class OverloadCons {
    public static void main(String[] args) {
        // Utworzenie pudełek za pomocą różnych konstruktorów
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        double vol;

        // Pobranie objętości pierwszego pudełka
        vol = mybox1.volume();
    }
}
```

```

System.out.println("Objętość mybox1 wynosi " + vol);

// Pobranie objętości drugiego pudełka
vol = mybox2.volume();
System.out.println("Objętość mybox2 wynosi " + vol);

// Pobranie objętości sześcianu
vol = mycube.volume();
System.out.println("Objętość mycube wynosi " + vol);
}
}

```

Wynik działania programu jest następujący.

```

Objętość mybox1 wynosi 3000.0
Objętość mybox2 wynosi -1.0
Objętość mycube wynosi 343.0

```

Jak łatwo zauważyć, na podstawie argumentów podanych w operatorze `new` został wybrany odpowiedni przeciążony konstruktor.

Obiekty jako parametry

Do tej pory jako parametrów używaliśmy jedynie typów prostych. Często jednak do metod przekazuje się obiekty. Rozważmy następujący program.

```

// R07\PassOb.java

// Do metod można przekazywać obiekty
class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // Zwraca true, jeśli obiekt o jest równy aktualnemu obiektowi
    boolean equalTo(Test o) {
        if(o.a == a && o.b == b) return true;
        else return false;
    }
}

class PassOb {
    public static void main(String[] args) {
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);

        System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
    }
}

```

Program generuje następujące wyniki.

```

ob1 == ob2: true
ob1 == ob3: false

```

Metoda `equalTo()` klasy `Test` porównuje dwa obiekty pod względem równości i zwraca wynik takiego porównania. Innymi słowy, porównuje aktualny obiekt z przekazanym jako parametr. Jeśli obiekty zawierają te same wartości, metoda zwraca wartość `true`. W przeciwnym razie zwraca wartość `false`. Zauważ, że jako typ parametru metody `equalTo()` pojawia się nazwa klasy `Test`. Choć klasa ta jest definiowana bezpośrednio w programie, Java traktuje ją dokładnie tak samo jak typy wbudowane.

Bardzo często parametry obiektowe pojawiają się w konstruktorach. Na przykład zależy nam na tym, by nowy obiekt został początkowo ustawiony na takie same wartości jak inny obiekt. Poniższa wersja klasy Box dopuszcza inicjalizację nowego obiektu Box innym obiektem tej samej klasy.

// R07\OverloadCons2.java

// Jeden obiekt Box może posłużyć do inicjalizacji innego obiektu Box

```
class Box {
    double width;
    double height;
    double depth;

    // Tworzenie klonu obiektu
    Box(Box ob) { // Przekazanie obiektu do konstruktora
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // Konstruktor używany, gdy podano wszystkie wymiary
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // Konstruktor używany przy braku wymiarów
    Box() {
        width = -1; // Wartość -1
        height = -1; // wskazuje
        depth = -1; // niezainicjalizowane pudełko
    }

    // Konstruktor używany do tworzenia sześcianów
    Box(double len) {
        width = height = depth = len;
    }

    // Oblicz i zwróć objętość
    double volume() {
        return width * height * depth;
    }
}

class OverloadCons2 {
    public static void main(String[] args) {
        // Tworzenie pudełek za pomocą różnych konstruktorów
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        Box myclone = new Box(mybox1); // Tworzenie kopii obiektu mybox1

        double vol;

        // Pobranie objętości pierwszego pudełka
        vol = mybox1.volume();
        System.out.println("Objętość mybox1 wynosi " + vol);

        // Pobranie objętości drugiego pudełka
        vol = mybox2.volume();
        System.out.println("Objętość mybox2 wynosi " + vol);

        // Pobranie objętości sześcianu
        vol = mycube.volume();
        System.out.println("Objętość mycube wynosi " + vol);
    }
}
```

```
// Pobranie objętości klonu
vol = myclone.volume();
System.out.println("Objętość clone wynosi " + vol);
}
}
```

Każdy programista, który zaczyna tworzyć własne klasy, szybko odkrywa, że definiowanie wielu form konstruktorów nierzadko decyduje o efektywności obiektów i wygodzie ich używania.

Dokładniejsze omówienie przekazywania argumentów

Ogólnie w językach programowania istnieją dwa sposoby przekazywania wartości do podprogramów. Pierwszy nosi nazwę **przekazywania przez wartość**. Powoduje to skopiowanie **wartości** argumentu do zmiennej będącej parametrem podprogramu. Z tego względu modyfikacje parametru wykonane w podprogramie nie mają wpływu na argument. Drugi sposób to **przekazywanie przez referencję**. W tym podejściu do zmiennej parametru wpisywana jest referencja do argumentu (zamiast wartości argumentu). Podprogram używa referencji do odczytania wartości przekazanego argumentu. Z tego względu podprogram może zmienić zawartość przekazanego argumentu. Java stosuje oba sposoby w zależności od tego, co jest przekazywane.

W Javie typy proste są przekazywane do metod przez wartość, więc zmiana parametru w metodzie nie wpływa na zawartość argumentu poza nią. Rozważmy następujący program.

```
// R07\CallByValue.java
// Typy proste są przekazywane przez wartość
class Test {
    void meth(int i, int j) {
        i *= 2;
        j /= 2;
    }
}

class CallByValue {
    public static void main(String[] args) {
        Test ob = new Test();

        int a = 15, b = 20;

        System.out.println("a i b przed wywołaniem: " +
            a + " " + b);

        ob.meth(a, b);

        System.out.println("a i b po wywołaniu: " +
            a + " " + b);
    }
}
```

Wyniki działania programu są następujące.

```
a i b przed wywołaniem: 15 20
a i b po wywołaniu: 15 20
```

Operacje wykonywane wewnątrz metody `meth()` w żaden sposób nie wpłynęły na wartości zmiennych `a` i `b` użytych w wywołaniu metody. Nie zostały im przypisane wartości 30 i 10.

Gdy do metody przekazuje się obiekt, sytuacja ulega dramatycznej zmianie, ponieważ przekazywanie obiektów odbywa się przez referencję. Pamiętaj, że zmienna typu klasowego przechowuje tak naprawdę tylko referencję do obiektu. Przekazanie tej referencji do metody powoduje, że parametr będzie wskazywał na ten sam obiekt, na który wskazywał argument w wywołaniu. Oznacza to, że zmiana obiektu wewnątrz metody *wpłyne* na obiekt przekazany jako argument. Rozważmy następujący program.

```
// R07\PassObjRef.java
// Obiekty są przekazywane przez referencję

class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // Przekazanie obiektu
    void meth(Test o) {
        o.a *= 2;
        o.b /= 2;
    }
}

class PassObjRef {
    public static void main(String[] args) {
        Test ob = new Test(15, 20);

        System.out.println("ob.a i ob.b przed wywołaniem: " +
            ob.a + " " + ob.b);

        ob.meth(ob);

        System.out.println("ob.a i ob.b po wywołaniu: " +
            ob.a + " " + ob.b);
    }
}
```

Program po wykonaniu generuje następujące wyniki.

```
ob.a i ob.b przed wywołaniem: 15 20
ob.a i ob.b po wywołaniu: 30 10
```

Jak łatwo zauważyć, instrukcje zawarte wewnątrz metody `meth()` spowodowały modyfikację obiektu przekazanego jako argument.



Pamiętaj!

Gdy dochodzi do przekazywania referencji do obiektu w wywołaniu metody, sama referencja jest przekazywana przez wartość. Ponieważ jednak wskazuje ona pewien obiekt, jej kopia zapisana do parametru także dotyczy tego obiektu.

Zwracanie obiektów

Metoda może zwrócić dowolny typ danych, w tym typ klasowy. Na przykład w poniższym przykładzie metoda `incrByTen()` zwraca obiekt, którego wartość `a` została zwiększona o 10 w stosunku do obiektu, dla którego metoda została wywołana.

```
// R07\RetOb.java
// Zwrócenie obiektu
class Test {
    int a;

    Test(int i) {
        a = i;
    }

    Test incrByTen() {
        Test temp = new Test(a+10);
        return temp;
    }
}
```



```

class RetOb {
    public static void main(String[] args) {
        Test ob1 = new Test(2);
        Test ob2;

        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);

        ob2 = ob2.incrByTen();
        System.out.println("ob2.a po drugim zwiększeniu: "
                           + ob2.a);
    }
}

```

Program generuje następujące wyniki.

```

ob1.a: 2
ob2.a: 12
ob2.a po drugim zwiększeniu: 22

```

W każdym wywołaniu metody `incrByTen()` powstaje nowy obiekt i zwracana jest referencja do niego.

Po analizie powyższego programu warto zwrócić uwagę na istotny fakt: ponieważ wszystkie obiekty są alokowane dynamicznie za pomocą operatora `new`, nie trzeba przejmować się utratą zasięgu obiektu spowodowaną zakończeniem metody, w której został zdefiniowany. Obiekt będzie istniał tak długo, jak istnieje gdzieś chociaż jedna referencja do niego. Jeśli nie będzie żadnych referencji, mechanizm odzyskiwania pamięci zniszczy taki obiekt.

Rekurencja

Java obsługuje **rekurencję**. Rekurencja polega na odwołaniu się do siebie samego w trakcie wykonywania obliczeń. W kontekście programowania rekurencja oznacza, że metoda może wywołać samą siebie. Taką metodę nazywa się **metodą rekurencyjną**.

Klasycznym przykładem rekurencji jest obliczanie silni z liczby. Silnia z liczby N to wynik mnożenia wszystkich liczb całkowitych od 1 do N . Na przykład silnia z 3 (oznaczana jako $3!$) to wynik działania $1 \times 2 \times 3 = 6$. Poniżej znajduje się metoda wykorzystująca rekurencję do obliczenia silni.

```

// R07\Recursion.java
// Prosty przykład rekurencji
class Factorial {
    // To jest metoda rekurencyjna
    int fact(int n) {
        int result;

        if(n==1) return 1;
        result = fact(n-1) * n;
        return result;
    }
}

class Recursion {
    public static void main(String[] args) {
        Factorial f = new Factorial();

        System.out.println("3! wynosi " + f.fact(3));
        System.out.println("4! wynosi " + f.fact(4));
        System.out.println("5! wynosi " + f.fact(5));
    }
}

```

Wyniki działania programu są następujące.

```

3! wynosi 6
4! wynosi 24
5! wynosi 120

```

Jeśli wcześniej nie korzystało się z funkcji rekurencyjnych, zapis metody `fact()` może wydawać się nieco dziwny. Oto sposób działania metody. Gdy argumentem metody `fact()` będzie wartość 1, metoda zwróci wartość 1. W przeciwnym razie zwróci wynik działania `fact(n-1)*n`. W celu obliczenia tego wyrażenia metoda jest wywoływana ponownie z argumentem `n-1`. Cały proces powtarza się aż do momentu, w którym `n` będzie równe 1 — wtedy nastąpi zwrócenie wszystkich wartości.

Aby lepiej zrozumieć sposób działania metody `fact()`, wykorzystajmy prosty przykład. W momencie obliczenia silni dla wartości 3 wywołanie `fact(3)` spowoduje przeprowadzenie drugiego wywołania z argumentem równym 2. To wywołanie spowoduje ponowne wywołanie metody `fact()`, ale z argumentem 1. Wynikiem `fact(1)` jest wartość 1, która następnie zostanie pomnożona przez wartość 2 (wartość `n` w drugim wywołaniu). Wynik (wartość 2) zostanie zwrócony do oryginalnego wywołania metody i pomnożony przez 3 (oryginalna wartość `n`). Spowoduje to zwrócenie wartości 6. Warto umieścić w tej metodzie wywołanie metody `println()`, aby prześledzić pośrednie odpowiedzi na poszczególnych poziomach rekurencji.

Gdy metoda wywołuje samą siebie, na stosie alokowane są nowe lokalne zmienne i parametry, a kod metody jest wykonywany od początku z nowymi wartościami. Przy powrocie stare zmienne lokalne i parametry są usuwane ze stosu oraz dochodzi do wznowienia działania metody, która dokonała wywołania rekurencyjnego. Metody rekurencyjne przypominają w działaniu rozkładaną lunetę (najpierw są rozwijane, a następnie zwijane).

Wersje rekurencyjne wielu algorytmów mogą działać nieco wolniej od ich odpowiedników iteracyjnych, ponieważ dodatkowe wywołania metod generują dodatkowe koszty. Bardzo zaawansowane metody rekurencyjne mogą doprowadzić do przepełnienia stosu. Wynika to z faktu, iż każde kolejne rekurencyjne wywołanie metody powoduje alokację miejsca na zmienne lokalne. Jeśli dojdzie do przepełnienia stosu, system wykonawczy Javy zgłosi wyjątek. Na ogół jednak nie trzeba zaprzętać sobie głowy tą kwestią, gdyż w zasadzie dotyczy ona tylko błędnie napisanych metod rekurencyjnych.

Główna zaleta metod rekurencyjnych polega na tym, iż dzięki nim można znacznie łatwiej zapisać niektóre algorytmy, których wersje iteracyjne są bardzo złożone. Na przykład algorytm QuickSort raczej trudno zaimplementować w sposób iteracyjny. Pewne problemy, szczególnie te związane ze sztuczną inteligencją, na ogół prowadzą do rozwiązań rekurencyjnych. Co więcej, niektórym osobom łatwiej zrozumieć algorytm rekurencyjny niż iteracyjny.

Gdy pisze się metody rekurencyjne, musi w nich wystąpić instrukcja warunkowa `if`, która spowoduje, iż po spełnieniu pewnych warunków nie dojdzie do wykonania kroku rekurencyjnego. Jeśli się nie doda takiego warunku, nigdy nie dojdzie do powrotu. Jest to typowy błąd występujący przy pisaniu algorytmów rekurencyjnych. Warto w trakcie testowania dodać do metody rekurencyjnej instrukcję `println()`, aby móc śledzić, co dzieje się w metodzie.

Oto kolejny przykład zastosowania rekurencji. Metoda `printArray()` wyświetla i pierwszych elementów tablicy `values`.

// R07\Recursion2.java

// Kolejny przykład obrazujący rekurencję

```
class RecTest {
    int[] values;

    RecTest(int i) {
        values = new int[i];
    }

    // Rekurencyjne wyświetlanie tablicy
    void printArray(int i) {
        if(i==0) return;
        else printArray(i-1);
        System.out.println "[" + (i-1) + " ] " + values[i-1];
    }
}

class Recursion2 {
    public static void main(String[] args) {
        RecTest ob = new RecTest(10);
        int i;
```

```

    for(i=0; i<10; i++) ob.values[i] = i;

    ob.printArray(10);
}
}

```

Wynik działania programu jest następujący.

```

[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9

```

Wprowadzenie do kontroli dostępu

Hermetyzacja łączy dane z kodem, który je modyfikuje. Hermetyzacja obejmuje też jeszcze jeden ważny element — **kontrolę dostępu**. Dzięki hermetyzacji można określić, które części programu powinny mieć dostęp do składowych klasy. W ten sposób można łatwo zapobiec nadużyciom. Udostępniając dane tylko za pomocą dobrze zdefiniowanego interfejsu (metod), zapobiega się niewłaściwemu użyciu lub zmodyfikowaniu danych. Dobrze napisana klasa jest jak czarna skrzynka, która może być używana, ale jej wnętrze jest chronione przed niepowołanym dostępem. Prezentowane do tej pory klasy nie do końca korzystały z tych rozwiązań. Na przykład klasa `Stack` z końca poprzedniego rozdziału udostępnia metody `pop()` i `push()` stanowiące interfejs do sterowania stosem. Stosowanie tego interfejsu nie jest jednak w żaden sposób wymuszane. Kod innej klasy może obejść te metody i bezpośrednio zmienić zawartość tablicy. Niepowołane osoby mogą w ten sposób popsuć działanie stosu. W tym podrozdziale omówię mechanizm zapewniający kontrolowany dostęp do poszczególnych składowych klasy.

Sposób dostępu do składowej klasy określa **modyfikator dostępu** znajdujący się na samym początku deklaracji elementu. Java oferuje bogaty zestaw modyfikatorów. Pewne aspekty sterowania dostępem są związane z dziedziczeniem oraz pakietami. (**Pakiet** to, ogólnie rzecz biorąc, grupa klas). Modyfikatory z tej grupy zostaną omówione w dalszych rozdziałach. Tutaj zajmę się tylko dostępem z perspektywy pojedynczej klasy. Programista, który raz opanuje podstawy kontroli dostępu, bez trudu zrozumie pozostałe zagadnienia.



Uwaga

Wpływ na dostępność poszczególnych elementów kodu mogą mieć także moduły — możliwość wprowadzona do języka Java w JDK 9. Moduły zostaną przedstawione w rozdziale 16.

Modyfikatory dostępu w języku Java to `public`, `private` i `protected`. Dodatkowo istnieje jeszcze domyślny poziom dostępu. Modyfikator `protected` dotyczy dziedziczenia. Pozostałe modyfikatory zostały omówione poniżej.

Zacznę od zdefiniowania dostępu publicznego i prywatnego. Gdy składowa klasy stosuje modyfikator `public`, oznacza to, że jest dostępna dla dowolnego innego kodu. Gdy stosuje modyfikator `private`, oznacza to, że jest dostępna tylko dla kodu swojej klasy. W ten sposób łatwo wyjaśnić, dlaczego metoda `main()` zawsze musi być poprzedzona modyfikatorem `public` — ponieważ jest wywoływana przez zewnętrzny kod, w tym przypadku system wykonawczy Javy. Jeśli nie zastosuje się żadnego modyfikatora, składowa klasy jest dostępna publicznie dla wszystkich innych klas swojego pakietu, ale nie jest dostępna dla kodu z innych pakietów. (Pakiety zostaną omówione w rozdziale 9.).

W tworzonych do tej pory klasach wszystkie składowe używały domyślnego poziomu dostępu. Niestety, bardzo rzadko korzysta się z takiego rozwiązania. Najczęściej bardzo mocno ogranicza się dostęp do danych klasy — innymi słowy, wymusza się, by dane były dostępne jedynie za pomocą metod. Także niektóre metody pomocnicze są deklarowane jako prywatne składowe klasy.

Modyfikator dostępu zawsze znajduje się na samym początku specyfikacji składowej, czyli musi rozpoczynać instrukcję deklaracji. Oto przykład.

```
public int i;
private double j;
```

```
private int myMethod(int a, char b) { //...
```

Aby lepiej zrozumieć efekt działania modyfikatorów, rozważmy następujący program.

```
// R07\AccessTest.java
/* Program ilustruje różnicę między dostępem
   prywatnym i publicznym
 */
class Test {
    int a; // Dostęp domyślny
    public int b; // Dostęp publiczny
    private int c; // Dostęp prywatny

    // Metoda korzystająca ze zmiennej c
    void setc(int i) { // Ustawienie wartości zmiennej c
        c = i;
    }
    int getc() { // Pobranie wartości zmiennej c
        return c;
    }
}

class AccessTest {
    public static void main(String[] args) {
        Test ob = new Test();

        // Poniższe instrukcje są poprawne, zmienne a i b są dostępne publicznie
        ob.a = 10;
        ob.b = 20;

        // Wykonanie poniższej instrukcji spowodowałoby zgłoszenie błędu
        // ob.c = 100; // Błąd!

        // Dostęp do zmiennej c tylko przez metodę
        ob.setc(100); // OK

        System.out.println("a, b i c: " + ob.a + " " +
                           ob.b + " " + ob.getc());
    }
}
```

Wewnątrz klasy Test zmienna składowa a używa dostępu domyślnego, który w tym przykładzie jest równoważny dostępowi publicznemu. Zmiennej b zostaje jawnie przyporządkowany dostęp publiczny (public). Zmiennej c zostaje jawnie przyporządkowany dostęp prywatny (private). Oznacza to, że klasa AccessTest nie może bezpośrednio odwołać się do zmiennej składowej c. Z tego powodu poniższy wiersz jest opatrzony komentarzem.

```
// ob.c = 100; // Błąd!
```

Gdyby usunąć komentarz, w trakcie kompilacji zostałby zgłoszony błąd dostępu.

Aby przekonać się, jak stosować mechanizm kontroli dostępu w bardziej praktycznym przykładzie, przeanalizujmy poniższą, ulepszoną wersję klasy Stack z końca rozdziału 6.

```
// R07\Stack.java
/* Klasa definiuje stos liczb całkowitych mogący przechowywać do 10 wartości
 */
class Stack {
    /* Teraz zmienne stck i tos są prywatne. Oznacza to, że
       nie mogą zostać zmienione celowo lub przypadkowo
       w sposób zagrażający działaniu stosu
    */
    private int[] stck = new int[10];
```

```

private int tos;

// Inicjalizacja szczytu stosu
Stack() {
    tos = -1;
}

// Umieszczenie elementu na szczycie stosu
void push(int item) {
    if(tos==9)
        System.out.println("Stos jest pełny.");
    else
        stck[++tos] = item;
}

// Zdjęcie elementu ze szczytu stosu
int pop() {
    if(tos < 0) {
        System.out.println("Stos nie zawiera żadnych elementów.");
        return 0;
    }
    else
        return stck[tos--];
}
}

```

Teraz zarówno składowa `stck` (zawierająca stos), jak i zmienna `tos` (zawierająca indeks najwyższego elementu) zostały zdefiniowane jako składowe prywatne (`private`). Oznacza to, że nie można ich zmienić ani odczytać w sposób bezpośredni — trzeba skorzystać z metod `pop()` i `push()`. Takie rozwiązanie chroni na przykład zmienną `tos` przed ustawieniem wartości spoza zakresu tablicy przez inny fragment programu.

Poniższy program wykorzystuje ulepszoną wersję klasy `Stack`. Usuń komentarze z ostatnich wierszy, aby przekonać się, że zmienne składowe `tos` i `stck` naprawdę nie są dostępne poza klasą `Stack`.

```

// R07\TestStack.java

class TestStack {
    public static void main(String[] args) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        // Umieszczenie liczb na stosach
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        // Zdjęcie liczb ze stosów
        System.out.println("Stos w mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stos w mystack2:");

        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());

        // Poniższe instrukcje nie są poprawne
        // mystack1.tos = -2;
        // mystack2.stck[3] = 100;
    }
}

```

Choć na ogół to właśnie metody zapewniają dostęp do danych klasy, nie zawsze musi tak być. Jeżeli z pewnych powodów zmienna składowa powinna być publiczna, warto tak zrobić. Na przykład w większości prostych klas zdefiniowanych w tej książce zrezygnowano z kontroli dostępu, aby maksymalnie uprościć ich kod. W rzeczywistych zastosowaniach najlepiej jednak udostępniać dane tylko poprzez metody. Wróćmy do tego zagadnienia także w następnym rozdziale, ponieważ kontrola dostępu należy do najważniejszych aspektów dziedziczenia.

Składowe statyczne

Czasem zachodzi potrzeba zdefiniowania składowej klasy, która nie zależałaby od żadnego konkretnego obiektu tej klasy. W tradycyjnym podejściu składowa dostępna jest tylko w powiązaniu z konkretną instancją klasy (obiektem). W Javie można jednak utworzyć składową udostępnianą bez podawania konkretnego obiektu. Aby utworzyć taki element, trzeba poprzedzić jego deklarację słowem kluczowym `static`. Składową statyczną można odczytywać i modyfikować, zanim zostaną utworzone jakiegokolwiek obiekty klasy, i nie jest do tego potrzebna referencja do obiektu. Statyczne mogą być zarówno zmienne, jak i metody. Najlepszym przykładem składowej statycznej jest metoda `main()` — musi ona zostać zadeklarowana ze słowem `static`, ponieważ jest wywoływana przed utworzeniem jakichkolwiek obiektów.

Zmienne składowe zadeklarowane jako statyczne są tak naprawdę zmiennymi globalnymi. Gdy Java tworzy obiekt takiej klasy, jej zmienne statyczne nie są kopiowane. Wszystkie obiekty klasy współdzielą tę samą zmienną statyczną.

Metody zadeklarowane jako `static` mają kilka ograniczeń.

- Mogą wywoływać tylko inne metody statyczne tej samej klasy.
- Mają dostęp tylko do zmiennych statycznych tej samej klasy.
- Nie mogą w żaden sposób korzystać ze słów kluczowych `this` i `super`. (Słowo kluczowe `super` jest związane z dziedziczeniem i zostanie omówione w kolejnym rozdziale).

Jeżeli do inicjalizacji zmiennych `static` potrzebne są dodatkowe obliczenia, deklaruje się blok `static`, który zostanie wykonany dokładnie jeden raz w momencie pierwszego załadowania klasy. Poniższy przykład przedstawia klasę zawierającą metodę statyczną, kilka zmiennych statycznych i blok inicjujący `static`.

```
// R07\UseStatic.java
```

```
// Przykład statycznych zmiennych, metod i bloków
```

```
class UseStatic {
    static int a = 3;
    static int b;

    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }

    static {
        System.out.println("Inicjalizacja w bloku statycznym");
        b = a * 4;
    }

    public static void main(String[] args) {
        meth(42);
    }
}
```

W momencie załadowania klasy `UseStatic` zostaną wykonane wszystkie wyrażenia oznaczone jako część statyczna tej klasy. W pierwszym kroku zmiennej `a` zostanie przypisana wartość 3, następnie zostanie wykonany blok `static` (wyświetlenie komunikatu), a na końcu zmienna `b` przyjmie wartość `a * 4`, czyli 12. Wykonanie metody `main()` spowoduje wywołanie metody `meth()` i przekazanie 42 jako argumentu. Trzy wywołania metody `println()` otrzymują na wejściu odpowiednio dwie zmienne statyczne (`a` i `b`) oraz parametr `x`.

Wynik działania programu jest następujący.

```
Inicjalizacja w bloku statycznym
x = 42
a = 3
b = 12
```

Metody i zmienne statyczne poza klasą, w której zostały zdefiniowane, są używane niezależnie od obiektów tej klasy. Trzeba wtedy użyć nazwy samej klasy wraz z operatorem kropki. Na przykład wywołanie metody statycznej spoza jej klasy ma następującą ogólną postać.

```
nazwa-klassy.metoda()
```

Element *nazwa-klassy* to nazwa klasy, w której znajduje się wywoływana metoda statyczna. Format jest więc bardzo podobny do stosowanego dla metod niestatycznych, choć wtedy zamiast nazwy klasy występuje nazwa obiektu. Zmienne statyczne odczytuje się w identyczny sposób, używając nazwy klasy i operatora kropki przed nazwą zmiennej. W ten sposób Java implementuje łatwą do sterowania wersję globalnych metod i zmiennych.

Oto kolejny przykład. Metoda `main()` znajdująca się w osobnej klasie korzysta z metody statycznej `callme()` i zmiennej statycznej `b`.

```
// R07\StaticDemo.java

class StaticDemo {
    static int a = 42;
    static int b = 99;

    static void callme() {
        System.out.println("a = " + a);
    }
}

class StaticByName {
    public static void main(String[] args) {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
    }
}
```

Program generuje następujące wyniki.

```
a = 42
b = 99
```

Słowo kluczowe final

Zmienne można zadeklarować z użyciem słowa kluczowego `final`. Słowo kluczowe `final` wyklucza możliwość modyfikacji wartości tak zadeklarowanej zmiennej, czyli w praktyce służy do deklarowania stałych. Oznacza to również, że inicjalizacja zmiennej tego typu musi nastąpić na etapie deklaracji. Taka deklaracja może mieć jedną z dwóch form — możemy przypisać wartość zmiennej albo w momencie deklaracji, albo w ciele konstruktora. To pierwsze rozwiązanie jest prawdopodobnie najczęściej stosowane. Oto przykład.

```
// R07\listing_17.java

final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Inne części programu mogą stosować zmienne `FILE_NEW`, `FILE_OPEN` itd. bez obawy o to, iż zostaną one w pewnym momencie zmodyfikowane. W zasadzie zmienne te można traktować jako stałe. Zgodnie z powszechną konwencją programistyczną nazwy zmiennych tego typu zapisuje się wielkimi literami.

Słowo kluczowe `final` można stosować nie tylko dla pól, ale też dla parametrów metod i zmiennych lokalnych. Zadeklarowanie parametru metody z tym słowem zapobiega zmianom tego parametru w ciele metody. Zadeklarowanie zmiennej ze słowem `final` wyklucza możliwość przypisania wartości tej zmiennej więcej niż raz.

Słowo kluczowe `final` stosuje się również dla metod, ale wtedy jego znaczenie jest całkowicie inne niż dla zmiennych. Zastosowanie tego słowa kluczowego dla metod zostanie dokładniej omówione w następnym rozdziale przy wprowadzaniu dziedziczenia.

Powtórka z tablic

Tablice zostały w niniejszej książce wprowadzone bardzo wcześnie, jeszcze przed omówieniem klas. Skoro już znamy klasy, warto wspomnieć o jednej z cech tablic — tablice są implementowane jak obiekty. Z tego względu istnieje specjalny atrybut tablicy, z którego korzysta się niezmiernie często. Zmienna składowa `length` tablicy zawiera informację o liczbie elementów, które może przyjąć tablica. Wszystkie tablice mają tę zmienną i zawsze przechowuje ona rozmiar tablicy. Oto program wykorzystujący tę właściwość.

```
// R07\Length.java
// Program korzystający ze zmiennej length tablic
class Length {
    public static void main(String[] args) {
        int[] a1 = new int[10];
        int[] a2 = {3, 5, 7, 1, 8, 99, 44, -10};
        int[] a3 = {4, 3, 2, 1};

        System.out.println("rozmiar a1 wynosi " + a1.length);
        System.out.println("rozmiar a2 wynosi " + a2.length);
        System.out.println("rozmiar a3 wynosi " + a3.length);
    }
}
```

Program generuje następujące wyniki.

```
rozmiar a1 wynosi 10
rozmiar a2 wynosi 8
rozmiar a3 wynosi 4
```

Jak łatwo zauważyć, program wyświetla rozmiary poszczególnych tablic. Pamiętaj, że wartość zawarta w `length` nie ma nic wspólnego z liczbą elementów rzeczywiście znajdujących się w tablicy. Zmienna informuje jedynie o maksymalnej liczbie elementów, które mogą zostać umieszczone w tablicy.

Istnieje bardzo dużo zastosowań wartości ze zmiennej `length`. Poniżej znajduje się ulepszona wersja klasy `Stack`. Poprzednia wersja klasy zawsze tworzyła stos 10-elementowy. Nowa wersja potrafi utworzyć stos o dowolnym rozmiarze. Wartość `stack.length` stosujemy w celu uniemożliwienia przepełnienia stosu.

```
// R07\TestStack2.java
// Ulepszona klasa Stack wykorzystująca rozmiar tablicy
class Stack {
    private int[] stck;
    private int tos;

    // Alokacja i inicjalizacja stosu
    Stack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // Umieszczenie elementu na szczycie stosu
    void push(int item) {
        if(tos==stck.length-1) // Użycie zmiennej składowej length
            System.out.println("Stos jest pełny.");
        else
            stck[++tos] = item;
    }
}
```



```

// Zdjęcie elementu ze szczytu stosu
int pop() {
    if(tos < 0) {
        System.out.println("Stos nie zawiera żadnych elementów.");
        return 0;
    }
    else
        return stck[tos--];
}
}

class TestStack2 {
    public static void main(String[] args) {
        Stack mystack1 = new Stack(5);
        Stack mystack2 = new Stack(8);

        // Umieszczenie liczb na stosach
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=10; i<8; i++) mystack2.push(i);

        // Zdjęcie liczb ze stosów
        System.out.println("Stos w mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stos w mystack2:");
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}

```

Zauważ, że program tworzy dwa stosy: jeden pięcioelementowy, drugi ośmioelementowy. Ponieważ informacja o rozmiarze znajduje się w samej tablicy, łatwo wykonać stos o dowolnym rozmiarze.

Klasy zagnieżdżone i klasy wewnętrzne

Można zdefiniować klasę wewnątrz innej klasy. Wtedy klasę znajdującą się wewnątrz nazywamy **klasą zagnieżdżoną**. Zasięg zagnieżdżonej klasy jest ściśle związany z zasięgiem klasy zewnętrznej. Jeśli klasa B została zdefiniowana wewnątrz klasy A, klasa B nie może istnieć niezależnie od klasy A (poza tą klasą). Zagnieżdżona klasa ma dostęp do wszystkich składowych, także prywatnych, należących do klasy zewnętrznej. Klasa zewnętrzna nie ma dostępu do składowych klasy zagnieżdżonej. Klasa zagnieżdżona, którą zadeklarowano bezpośrednio w zasięgu klasy zewnętrznej, jest składową klasy zewnętrznej. Istnieje też możliwość deklarowania klas zagnieżdżonych w ramach bloków kodu.

Istnieją dwa rodzaje klas zagnieżdżonych: **statyczne** i **niestatyczne**. Statyczna klasa zagnieżdżona posiada na początku modyfikator `static`. Ponieważ jest klasą statyczną, dostęp do składowych klasy zewnętrznej musi się odbywać przez konkretną nazwę obiektu (nie można odwoływać się do zmiennych klasy zewnętrznej w sposób bezpośredni).

Drugim rodzajem klas zagnieżdżonych są klasy **wewnętrzne**. Klasa wewnętrzna to niestatyczna klasa zagnieżdżona. Ma ona pełny dostęp do wszystkich zmiennych i metod klasy zewnętrznej i może się do nich odnosić w sposób bezpośredni, podobnie jak metody klasy zewnętrznej.

Poniższy program ilustruje definiowanie i zastosowanie klas wewnętrznych. Klasa o nazwie `Outer` zawiera jedną zmienną składową `outer_x`, jedną metodę składową `test()` oraz definicję klasy wewnętrznej `Inner`.

```

// R07\InnerClassDemo.java

// Przykład klasy wewnętrznej
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
    }
}

```

```

        inner.display();
    }

    // To jest klasa wewnętrzna
    class Inner {
        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }
}

class InnerClassDemo {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();
    }
}

```

Wynik działania programu jest następujący.

```
display: outer_x = 100
```

W programie klasa wewnętrzna o nazwie `Inner` została zdefiniowana w zasięgu klasy `Outer`. Z tego powodu dowolny kod klasy `Inner` ma bezpośredni dostęp do zmiennej `outer_x`. Metoda `display()` znajduje się wewnątrz klasy `Inner` i powoduje przekazanie wartości zmiennej `outer_x` na standardowe wyjście. Metoda `main()` klasy `InnerClassDemo` tworzy instancję klasy `Outer` i wywołuje metodę `test()` dla nowego obiektu. Metoda `test()` tworzy instancję klasy `Inner` i wywołuje metodę `display()`.

Należy podkreślić istotny fakt: klasa `Inner` znana jest jedynie w kontekście klasy `Outer`. Próba utworzenia instancji klasy `Inner` w innym miejscu spowoduje, że kompilator Javy zgłosi błąd. Ogólnie rzecz biorąc, instancje klasy wewnętrznej są zazwyczaj tworzone przez kod umieszczony w zasięgu, w którym klasa ta została zdefiniowana, co pokazano na poniższym przykładzie.

Klasa wewnętrzna ma dostęp do wszystkich elementów klasy zewnętrznej, ale sytuacja odwrotna nie jest prawdziwa. Składowe klasy wewnętrznej znane są tylko w jej zasięgu i nie są dostępne dla klasy zewnętrznej. Oto przykład.

```

// R07\InnerClassDemo2.java

// Tego programu nie uda się skompilować
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // Klasa wewnętrzna
    class Inner {
        int y = 10; // y jest lokalne dla Inner

        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }

    void showy() {
        System.out.println(y); // Błąd, y nie jest znane!
    }
}

class InnerClassDemo2 {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();
    }
}

```

W powyższym kodzie zmienna `y` stanowi składową klasy `Inner` i jako taka nie jest widoczna poza tą klasą, zatem nie może zostać wyświetlona przez metodę `showy()`.

Choć skupiliśmy się przede wszystkim na zagnieżdżonych klasach deklarowanych wewnątrz zasięgu klasy zewnętrznej, tak naprawdę klasy wewnętrzne można definiować w dowolnym bloku kodu. Nic nie stoi na przeszkodzie, aby zdefiniować klasę wewnątrz bloku metody lub nawet wewnątrz bloku pętli `for` (patrz poniższy przykład).

```
// R07\InnerClassDemo3.java
// Definicja klasy wewnętrznej wewnątrz bloku pętli for
class Outer {
    int outer_x = 100;

    void test() {
        for(int i=0; i<10; i++) {
            class Inner {
                void display() {
                    System.out.println("display: outer_x = " + outer_x);
                }
            }
            Inner inner = new Inner();
            inner.display();
        }
    }
}

class InnerClassDemo3 {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Program generuje następujące wyniki.

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

Choć w tradycyjnym programowaniu klasy wewnętrzne stosuje się niezmiernie rzadko, są one bardzo pomocne przy obsłudze zdarzeń w apletach. Do tematu klas wewnętrznych powrócę w rozdziale 25., w którym pokażę, w jaki sposób tego rodzaju klasy upraszczają kod obsługi zdarzeń. Omówię tam również **anonimowe klasy wewnętrzne** — klasy wewnętrzne bez jawnie określonych nazw.

Ostatnia uwaga: klasy zagnieżdżone zostały dodane dopiero w Javie 1.1, więc nie można ich stosować w kodzie zgodnym z Javą 1.0.

Omówienie klasy `String`

Choć klasa `String` zostanie szczegółowo omówiona w części II książki, już tutaj potrzebne jest pewne wprowadzenie, gdyż będziemy dosyć intensywnie korzystać z tej klasy w przykładach pod koniec części I książki. Klasa `String` jest prawdopodobnie najczęściej stosowaną klasą języka Java. To o tyle naturalne, że łańcuchy stanowią bardzo ważny aspekt programowania.

Przed wszystkim trzeba zrozumieć, iż każdy tekst w kodzie programu zostanie zamieniony na obiekt typu `String`. Dotyczy to również stałych tekstowych. Oto przykład.

```
System.out.println("To także jest obiekt klasy String.");
```

Stała tekstowa "To także jest obiekt klasy String." jest obiektem klasy String.

Drugą bardzo istotną kwestią jest tak zwana niezmiennosc obiektów typu String. Gdy obiekt String zostanie utworzony, jego zawartość nie może zostać zmodyfikowana. Choć może się to wydawać dużym ograniczeniem, nie jest tak z dwóch powodów:

- Gdy modyfikuje się łańcuch znaków, zawsze można utworzyć nowy łańcuch zawierający wszystkie modyfikacje.
- Java definiuje klasy pomocnicze StringBuffer oraz StringBuilder, które umożliwiają modyfikację łańcuchów, więc wszystkie tradycyjne operacje na łańcuchach są dostępne również w Javie. (Klasy StringBuffer i StringBuilder są dokładnie omówione w części II książki).

Obiekty String tworzy się na wiele różnych sposobów. Najprościej użyć poniższego polecenia.

```
String myString = "to jest test";
```

Tak utworzony obiekt klasy String można wykorzystywać wszędzie tam, gdzie jest dopuszczalne stosowanie łańcuchów. Następujący wiersz spowoduje wyświetlenie zawartości zmiennej myString.

```
System.out.println(myString);
```

Java definiuje dla łańcuchów tylko jeden operator +, który służy do łączenia (konkatenacji) łańcuchów. Oto przykład.

```
String myString = "Lubię " + "Jawę.";
```

Powyższa instrukcja spowoduje przypisanie łańcucha "Lubię Jawę." zmiennej myString.

Następujący program obrazuje wykorzystanie konkatenacji łańcuchów znaków.

```
// R07\StringDemo.java
// Łączenie łańcuchów znaków
class StringDemo {
    public static void main(String[] args) {
        String strOb1 = "Pierwszy tekst";
        String strOb2 = "Drugi tekst";
        String strOb3 = strOb1 + " i " + strOb2;

        System.out.println(strOb1);
        System.out.println(strOb2);
        System.out.println(strOb3);
    }
}
```

Program po uruchomieniu generuje następujące komunikaty.

```
Pierwszy tekst
Drugi tekst
Pierwszy tekst i Drugi tekst
```

Klasa String zawiera wiele przydatnych metod. Tutaj przedstawię tylko kilka z nich. Do porównywania dwóch łańcuchów służy metoda equals(). Metoda length() zwraca długość łańcucha. Aby pobrać pojedynczy znak łańcucha z konkretnej pozycji, należy użyć metody charAt(). Ogólne postacie wspomnianych metod są następujące.

```
boolean equals(String obiekt)
int length()
char charAt(int indeks)
```

Poniżej znajduje się program wykorzystujący omawiane metody.

```
// R07\StringDemo2.java
// Przykład użycia kilku metod klasy String
class StringDemo2 {
    public static void main(String[] args) {
        String strOb1 = "Pierwszy tekst";
        String strOb2 = "Drugi tekst";
        String strOb3 = strOb1;
```

```

System.out.println("Długość str0b1: " +
                    str0b1.length());

System.out.println("Znak o indeksie 3 z str0b1: " +
                    str0b1.charAt(3));

if(str0b1.equals(str0b2))
    System.out.println("str0b1 == str0b2");
else
    System.out.println("str0b1 != str0b2");

if(str0b1.equals(str0b3))
    System.out.println("str0b1 == str0b3");
else
    System.out.println("str0b1 != str0b3");
}
}

```

Program generuje następujące wyniki.

```

Długość str0b1: 14
Znak o indeksie 3 z str0b1: r
str0b1 != str0b2
str0b1 == str0b3

```

Programista może oczywiście utworzyć tablicę łańcuchów, tak samo jak tworzy tablice elementów dowolnych innych typów. Oto przykład.

```

// R07\StringDemo3.java
// Tablica obiektów typu String
class StringDemo3 {
    public static void main(String[] args) {
        String[] str = { "jeden", "dwa", "trzy" };

        for(int i=0; i<str.length; i++)
            System.out.println("str[" + i + "]: " +
                               str[i]);
    }
}

```

Wynik działania programu jest następujący.

```

str[0]: jeden
str[1]: dwa
str[2]: trzy

```

W dalszej części rozdziału przekonamy się, że tablice łańcuchów odgrywają ważną rolę w wielu programach Javy.

Wykorzystanie argumentów wiersza poleceń

W pewnych sytuacjach zachodzi potrzeba przekazania do programu dodatkowych informacji w momencie jego uruchamiania. W Javie zadanie to polega na przekazaniu **argumentów wiersza poleceń** do metody `main()`. Argument wiersza poleceń to informacja występująca w wierszu poleceń tuż po nazwie uruchamianego programu. Odczytanie argumentów w języku Java jest bardzo proste — wystarczy sprawdzić tablicę obiektów `String` przekazywaną do metody `main()`. Poniższy program wyświetla wszystkie argumenty przekazane do niego w wierszu poleceń.

```

// R07\CommandLine.java
// Wyświetla wszystkie argumenty wiersza poleceń
class CommandLine {
    public static void main(String[] args) {
        for(int i=0; i<args.length; i++)
            System.out.println("args[" + i + "]: " +
                               args[i]);
    }
}

```

Uruchom program, używając następującego polecenia.

```
java CommandLine to jest test 100 -1
```

Wynik działania programu po wykonaniu tego zadania będzie następujący.

```
args[0]: to
args[1]: jest
args[2]: test
args[3]: 100
args[4]: -1
```



Pamiętaj

Wszystkie argumenty wiersza poleceń są przekazywane w formie łańcuchów. Wartości liczbowe należy więc ręcznie konwertować już w kodzie programu (patrz rozdział 19.).

Zmienna liczba argumentów

Nowoczesne wersje Javy oferują rozwiązanie ułatwiające tworzenie metod, które muszą przyjmować zmienną liczbę argumentów. **Zmienną liczbę argumentów** często nazywa się **varargs**. Metodę, która wykorzystuje zmienną liczbę argumentów, nazywa się **metodą zmiennesargumentową** lub **metodą varargs**.

Sytuacje, w których potrzeba zmiennej liczby argumentów, nie są rzadkością. Na przykład metoda otwierająca połączenie internetowe może albo otrzymywać nazwę użytkownika, hasło, nazwę pliku, protokół itp., albo stosować wartości domyślne dla argumentów, które nie zostały przekazane. W tej sytuacji najlepiej byłoby przekazywać do metody tylko te argumenty, których wartości domyślne nie pasują do bieżącego scenariusza. Innym przykładem jest metoda `printf()` stanowiąca część biblioteki wejścia-wyjścia. W rozdziale 21. pokażę, iż przyjmuje ona zmienną liczbę argumentów (format danych oraz wartości).

W początkowym okresie stosowania języka Java zmienną liczbę argumentów można było obsługiwać na dwa sposoby, z których żaden nie gwarantował należytej wygody. Jeśli maksymalna liczba argumentów była niewielka i dobrze znana, można było utworzyć przeciążone wersje metody z różną liczbą przyjmowanych argumentów. Niestety takie podejście, choć w miarę wygodne, mogło być stosowane tylko w niektórych sytuacjach.

Gdy maksymalna liczba argumentów była duża lub po prostu nieznana, stosowało się drugie rozwiązanie, wciąż spotykane w niektórych starych kodach, polegające na umieszczaniu argumentów w tablicy i przekazywaniu tej tablicy do metody. Takie podejście ilustruje poniższy program.

```
// R07\PassArray.java

// Użycie tablicy do przekazywania zmiennej liczby argumentów
// do metody. Jest to rozwiązanie stosowane przed wprowadzeniem
// metod o zmiennej liczbie argumentów
class PassArray {
    static void vaTest(int[] v) {
        System.out.print("Liczba argumentów: " + v.length +
            " Zawartość: ");

        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }

    public static void main(String[] args)
    {
        // Zauważ sposób tworzenia tablicy wymagany do
        // przechowywania argumentów
        int[] n1 = { 10 };
        int[] n2 = { 1, 2, 3 };
        int[] n3 = { };

        vaTest(n1); // 1 argument
        vaTest(n2); // 3 argumenty
```

```

    vaTest(n3); // Brak argumentów
}
}

```

Wynik działania programu jest następujący.

```

Liczba argumentów: 1 Zawartość: 10
Liczba argumentów: 3 Zawartość: 1 2 3
Liczba argumentów: 0 Zawartość:

```

W powyższym programie metoda `vaTest()` pobiera przekazane do niej argumenty za pośrednictwem tablicy `v`. Takie rozwiązanie stosowane w starszych programach Javy umożliwiało pobranie dowolnej liczby argumentów. Niestety wymagało ręcznego pakowania poszczególnych argumentów do tablicy przed wywołaniem metody `vaTest()`. Tworzenie tablicy przy każdym wywołaniu metody jest nie tylko żmudne, ale również łatwo popełnić błąd. Nowa cecha języka Java 5 dostarcza prostsze, lepsze rozwiązanie.

Zmienną liczbę argumentów określa się przez zastosowanie trzech kropek (...). Poniżej znajduje się początek definicji metody `vaTest()` w wersji `varargs`.

```
static void vaTest(int ... v) {
```

Powyższa składnia informuje kompilator, że metoda `vaTest()` może zostać wywołana z zerem lub większą liczbą argumentów. Parametr `v` jest niejawnie deklarowany jako tablica typu `int[]`. Z tego powodu wewnątrz metody `v` traktuje się go jak normalną tablicę. Poniżej znajduje się ten sam program co wcześniej, ale napisany przy użyciu nowego podejścia.

```
// R07\VarArgs.java
```

```
// Metoda o zmiennej liczbie argumentów
```

```
class VarArgs {

    // Metoda vaTest() jest typu varargs
    static void vaTest(int ... v) {
        System.out.print("Liczba argumentów: " + v.length +
            " Zawartość: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String[] args)
    {

        // Zauważ, że teraz metoda vaTest() jest
        // po prostu wywoływana ze zmienną liczbą argumentów
        vaTest(10);           // 1 argument
        vaTest(1, 2, 3);      // 3 argumenty
        vaTest();             // Brak argumentów
    }
}

```

Wynik działania powyższego programu jest dokładnie taki sam jak poprzedniego programu.

W programie warto zauważyć dwie istotne kwestie. Po pierwsze, we wnętrzu metody `vaTest()` zmienna `v` niczym nie różni się od tradycyjnej tablicy. To dość naturalne — w końcu zmienna `v` jest tablicą. Składnia ... po prostu informuje kompilator, aby wszystkie argumenty przekazane do metody umieścić w kolejnych elementach tablicy `v`. Po drugie, wewnątrz metody `main()` metoda `vaTest()` jest wywoływana z różną liczbą argumentów, a nawet bez żadnych argumentów. Programista nie musi zajmować się umieszczaniem argumentów w tablicy. Brak argumentów oznacza tablice o zerowej długości.

Oprócz parametrów ze zmienną liczbą argumentów metoda może otrzymywać także tradycyjne parametry. Wymaga to jednak umieszczenia parametru o zmiennej liczbie argumentów na samym końcu parametrów metody. Następująca deklaracja metody jest w pełni poprawna.

```
int doIt(int a, int b, double c, int ... vals) {
```

Pierwsze trzy argumenty przekazane do metody `doIt()` znajdują się w trzech pierwszych parametrach. Wszystkie pozostałe argumenty zostają umieszczone w tablicy `vals`.

Ponieważ parametr o zmiennej liczbie argumentów musi pojawić się jako ostatni, poniższa deklaracja nie jest poprawna.

```
int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Błąd!
```

Próba zadeklarowania tradycyjnego parametru po zmiennej liczbie argumentów zawsze spowoduje zgłoszenie błędu przez kompilator.

Co więcej, jedna metoda może posiadać tylko jeden parametr o zmiennej liczbie argumentów. Z tego powodu następująca deklaracja nie jest poprawna.

```
int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Błąd!
```

Próba zadeklarowania drugiego parametru o zmiennej liczbie argumentów zawsze spowoduje zgłoszenie błędu przez kompilator.

Poniżej znajduje się zmodyfikowana wersja metody `vaTest()`, która przyjmuje zarówno tradycyjny parametr, jak i zmienną liczbę argumentów.

```
// R07\VarArgs2.java
// Metoda typu varargs z tradycyjnymi parametrami
class VarArgs2 {

    // Tutaj msg to tradycyjny parametr, a v to
    // parametr typu varargs
    static void vaTest(String msg, int ... v) {
        System.out.print(msg + v.length +
                          " Zawartość: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String[] args)
    {
        vaTest("Jeden parametr typu vararg: ", 10);
        vaTest("Trzy parametry typu vararg: ", 1, 2, 3);
        vaTest("Brak parametrów typu vararg: ");
    }
}
```

Wynik działania programu jest następujący.

Jeden parametr typu vararg: 1 Zawartość: 10

Trzy parametry typu vararg: 3 Zawartość: 1 2 3

Brak parametrów typu vararg: 0 Zawartość:

Przeciążanie metod o zmiennej liczbie argumentów

Metody otrzymujące zmienną liczbę argumentów można przeciążać. Poniższa klasa posiada trzy przeciążone wersje metody `vaTest()`.

```
// R07\VarArgs3.java
// Metody typu vararg i przeciążanie
class VarArgs3 {

    static void vaTest(int ... v) {
        System.out.print("vaTest(int ...): " +
                          "Liczba argumentów: " + v.length +
                          " Zawartość: ");

        for(int x : v)
            System.out.print(x + " ");
    }
}
```



```

    System.out.println();
}

static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
        "Liczba argumentów: " + v.length +
        " Zawartość: ");

    for(boolean x : v)
        System.out.print(x + " ");

    System.out.println();
}

static void vaTest(String msg, int ... v) {
    System.out.print("vaTest(String, int ...): " +
        msg + v.length +
        " Zawartość: ");

    for(int x : v)
        System.out.print(x + " ");

    System.out.println();
}

public static void main(String[] args)
{
    vaTest(1, 2, 3);
    vaTest("Testowanie: ", 10, 20);
    vaTest(true, false, false);
}

```

Wynik działania programu jest następujący.

```

vaTest(int ...): Liczba argumentów: 3 Zawartość: 1 2 3
vaTest(String, int ...): Testowanie: 2 Zawartość: 10 20
vaTest(boolean ...) Liczba argumentów: 3 Zawartość: true false false

```

Program ilustruje oba dopuszczalne sposoby przeciążania metod o zmiennej liczbie argumentów. Pierwszy sposób to zróżnicowanie typów zmiennego parametru. Dotyczy to przypadków `vaTest(int ...)` oraz `vaTest(boolean ...)`. Pamiętaj, że składnia `...` powoduje traktowanie parametru jak tablicy określonego typu. Z tego powodu podobnie jak można tworzyć przeciążone metody o różnych typach tablic, można również tworzyć parametry typu `vararg` różnych typów. Java używa tego typu do rozróżnienia, którą wersję metody należy wywołać.

Drugi sposób przeciążania polega na dodaniu tradycyjnego parametru (przypadek `vaTest(String, int...)`). W takiej sytuacji do rozróżnienia wersji metody Java używa zarówno tradycyjnego parametru, jak i parametru o zmiennej liczbie argumentów.



Uwaga

Metoda `varargs` może być przeciążona także przez metodę pozbawioną parametrów wieloargumentowych. Na przykład metoda `vaTest(int x)` jest prawidłowym przeciążeniem metody `vaTest()` (patrz program w kolejnym punkcie). Wspomniana wersja zostanie wywołana tylko w przypadku przekazania pojedynczego argumentu typu `int`. Przekazanie co najmniej dwóch argumentów typu `int` spowoduje wykonanie wersji `varargs`, czyli `vaTest(int...v)`.

Zmienna liczba argumentów i niejednoznaczności

Gdy korzysta się z przeciążania metod o zmiennej liczbie argumentów, czasem mogą pojawić się nieoczekiwane wyniki. Wynikają one z niejednoznaczności wywołań tak przeciążonej metody. Rozważmy następujący program.

```
// R07\VarArgs4.java
// Zmienna liczba argumentów, przeciążanie i niejednoznaczność
//
// Program zawiera błąd, więc nie zostanie skompilowany!
class VarArgs4 {

    static void vaTest(int ... v) {
        System.out.print("vaTest(Integer ...): " +
            "Liczba argumentów: " + v.length +
            " Zawartość: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    static void vaTest(boolean ... v) {
        System.out.print("vaTest(boolean ...) " +
            "Liczba argumentów: " + v.length +
            " Zawartość: ");

        for(boolean x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String[] args)
    {
        vaTest(1, 2, 3); // OK
        vaTest(true, false, false); // OK

        vaTest(); // Błąd: niejednoznaczność!
    }
}
```

W programie przeciążone wersje metody `vaTest()` zostały napisane poprawnie. Niestety, programu nie uda się skompilować z powodu następującego wywołania.

```
vaTest(); // Błąd: niejednoznaczność!
```

Ponieważ parametr wieloargumentowy nie wymaga żadnych argumentów, powyższe wywołanie może dotyczyć zarówno metody `vaTest(int ...)`, jak i metody `vaTest(boolean ...)`. Obie wersje są równie poprawne, więc powstaje niejednoznaczność.

Oto inny przykład niejednoznaczności. Poniższe wersje metody `vaTest()` także mogą prowadzić do niejednoznaczności, choć jedna z wersji stosuje tradycyjny parametr.

```
static void vaTest(int ... v) { // ...

static void vaTest(int n, int ... v) { // ...
```

Choć lista parametrów obu wersji jest różna, kompilator nie potrafi wybrać odpowiedniej wersji metody dla poniższego wywołania.

```
vaText(1);
```

Przedstawione wywołanie pasuje zarówno do `vaTest(int ...)`, jak i do `vaTest(int, int ...)`. Kompilator nie potrafi rozstrzygnąć, która wersja jest poprawna — istnieje niejednoznaczność.

Ze względu na tego rodzaju problemy warto czasem poświęcić przeciążanie i po prostu zastosować dwie różne nazwy metod. Co więcej, czasami niejednoznaczności odsłaniają błędy koncepcyjne w kodzie programu, które warto poprawić.

Stosowanie wnioskowania typów zmiennych lokalnych z typami referencyjnymi

Jak już wiesz z rozdziału 3., od JDK 10 język Java udostępnia mechanizm wnioskowania typów zmiennych lokalnych. Zapewne pamiętasz, że podczas korzystania z wnioskowania typów zmiennych lokalnych typ zmiennej jest określany jako `var`, a sama zmienna musi zostać zainicjalizowana. We wcześniejszych przykładach przedstawiłem stosowanie tego mechanizmu z typami prostymi i referencyjnymi. Właściwie mechanizm ten jest stosowany przede wszystkim w zmiennych typów referencyjnych. Poniższy przykład przedstawia deklarację zmiennej `myStr` typu `String`:

```
var myStr = "To jest łańcuch";
```

Ponieważ inicjalizatorem jest łańcuch zapisany pomiędzy znakami cudzysłowu, wywnioskowanym typem będzie `String`.

Jak wyjaśniłem w rozdziale 3., jedną z korzyści stosowania mechanizmu wnioskowania typów zmiennych lokalnych jest skracanie i upraszczanie kodu, przy czym najłatwiej to zauważyć właśnie w przypadku stosowania typów referencyjnych.

Wynika to z faktu, że wiele klas w Javie ma długie nazwy. Na przykład w rozdziale 13. poznasz klasę `FileInputStream`, służącą do otwierania pliku w celu wczytywania danych. W przeszłości, gdy konieczne było stosowanie tradycyjnych deklaracji, zmienną tego typu trzeba było deklarować i inicjalizować w następujący sposób:

```
FileInputStream fin = new FileInputStream("test.txt");
```

Obecnie, w przypadku stosowania `var`, deklarację tę możemy zapisać tak:

```
var fin = new FileInputStream("test.txt");
```

Na podstawie takiej deklaracji Java wywnioskuje, że zmienna `fin` jest typu `FileInputStream`, gdyż taki jest typ użytego inicjalizatora. Nie trzeba jawnie powtarzać nazwy typu. W efekcie ta druga deklaracja zmiennej `fin` jest znacząco krótsza niż w przypadku stosowania tradycyjnego zapisu. Jak zatem widać, użycie `var` upraszcza deklarację. Ta zaleta mechanizmu wnioskowania typów zmiennych lokalnych staje się jeszcze wyraźniejsza w przypadku bardziej złożonych deklaracji, takich jak deklaracje korzystające z typów sparametryzowanych. Ogólnie rzecz ujmując, ta cecha mechanizmu wnioskowania typów zmiennych lokalnych, która umożliwia upraszczanie kodu, eliminuje w pewnym stopniu konieczność wpisywania w kodzie nazw typów.

Oczywiście mechanizm wnioskowania typów zmiennych lokalnych w celu upraszczania kodu musi być stosowany rozważnie, by nie doprowadził do pogorszenia czytelności kodu i ukrycia jego znaczenia. W ramach przykładu przyjrzymy się następującej deklaracji:

```
var x = o.getNext();
```

W tym przypadku osoba analizująca kod może nie być w stanie bezproblemowo określić, jakiego typu będzie zmienna `x`. A zatem, mówiąc ogólnie, wnioskowanie typów zmiennych lokalnych jest mechanizmem, którego należy używać mądrze.

Jak łatwo się domyślić, mechanizmu wnioskowania typów zmiennych lokalnych można także używać w przypadku korzystania z własnych klas, co pokazałem na kolejnym przykładzie. Poniższy program definiuje klasę `MyClass`, a następnie korzysta z mechanizmu wnioskowania typów zmiennych lokalnych w celu zadeklarowania i zainicjalizowania obiektu tej klasy:

```
// R07\RefVarDemo.java
```

```
// Użycie wnioskowania typów zmiennych lokalnych z klasą zdefiniowaną przez programistę
```

```
class MyClass {
    private int i;

    MyClass(int k) { i = k; }

    int geti() { return i; }
    void seti(int k) { if(k >= 0) i = k; }
}
```

```
class RefVarDemo {  
    public static void main(String[] args) {  
        var mc = new MyClass(10); // Zwróć tu uwagę na użycie var  
  
        System.out.println("Wartość i w mc wynosi " + mc.geti());  
        mc.seti(19);  
        System.out.println("A teraz wartość i w mc wynosi " + mc.geti());  
    }  
}
```

Wykonanie tego programu zwróci następujące wyniki:

```
Wartość i w mc wynosi 10  
A teraz wartość i w mc wynosi 19
```

W tym przykładzie zwróć szczególną uwagę na następujący wiersz:

```
var mc = new MyClass(10); // Zwróć tu uwagę na użycie var
```

Wywnioskowanym typem zmiennej `mc` będzie tu `MyClass`, gdyż inicjalizatorem jest nowy obiekt tej klasy.

Jak już zaznaczyłem wcześniej, z myślą o czytelnikach korzystających ze starszych wersji środowiska Javy, które nie obsługują mechanizmu wnioskowania typów zmiennych lokalnych, nie będę z niego korzystał w przykładach zamieszczonych w dalszej części książki. Dzięki temu większość czytelników będzie mogła bez problemu kompilować i wykonywać te przykłady.

Skorowidz

A

- abstrakcja, 48, 189
- adnotacja, 273
 - @Deprecated, 283
 - @Documented, 282
 - @FunctionalInterface, 284
 - @Inherited, 283
 - @MyAnno, 279
 - @Override, 283
 - @Repeatable, 288
 - @Retention, 282
 - @SafeVarargs, 284
 - @SuppressWarnings, 284
 - @Target, 283
- adnotacje
 - jednoelementowe, 281
 - odczytywanie, 275, 278
 - ograniczenia, 289
 - powtarzalne, 288
 - tworzenie, 273
 - typów, 284
 - wartości domyślne, 279
 - wbudowane, 282
 - znacznikowe, 280
- adres
 - internetowy, 696
 - URL, 1017, 1031
- akceleratorzy, 983, 984
- akcja, 993
- algorytmy, 503
 - klasy Collections, 545–549
- analizator leksykalny, 568
- analizowanie dat i godzin, 928
- anonimowe klasy wewnętrzne, 162, 744
- API, 711, 714
 - dat i czasu, 924
 - strumieni, 884, 906
- aplet, 35
- aplikacje
 - korzystające z klasy Frame, 753
 - korzystające z usług, 394
 - typu klient-serwer, 918–920
- architektura MVC, 935
- argumenty, 133, 136
 - wieloznaczne, 325
 - ograniczone, 328, 329
 - wiersza poleceń, 164
- ARM, automatic resource management, 219, 303, 499
- asercje, 309
 - opcje włączania i wyłączania, 311
- asynchroniczne wykonywanie zadań, 880
- automatyczne
 - opakowywanie typów, 270
 - logicznych, 272
 - prosty, 269
 - w wyrażeniach, 270
 - zapobieganie błędom, 272
 - znakowych, 272
 - rozpakowywanie, 270
 - rozszerzanie typów, 76
 - zamykanie pliku, 303
 - zarządzanie zasobami, ARM, 219, 303, 499
 - zwalnianie zasobu, 232
- AWT, Abstract Window Toolkit, 746, 771, 733, 933

B

barwa, 759
 bezpieczeństwo, 36
 typów, 320, 325
 białe znaki, 409
 biblioteka Swing, 733, 933
 dołączany wygląd i sposób obsługi, 934
 drzewo elementów, 969, 972
 klasy, 949
 klasy systemu menu, 976
 komponenty, 934, 936
 kontenery, 936
 lekkie, 937
 najwyższego poziomu, 937
 listy kombinowane, 967, 969
 menu główne, 979
 obsługa PLAF, 934
 obsługa zdarzeń, 941
 obszar rysowania, 945
 pakiety, 938
 paski przewijania, 964
 pola wyboru, 957
 przyciski, 953
 przyciski opcji, 958, 960
 rysowanie, 944, 946
 zakładki, 962
 biblioteki, 62
 blok
 kodu, 59, 103
 finally, 218, 227, 300
 try, 218, 300
 try-catch, 299
 tekstu, 399, 408
 blokada, 865
 wielobieżna, 866
 wzajemna, deadlock, 252
 blokowe wyrażenia lambda, 355
 błędy niejednoznaczności, 347
 bufor, 667

C

cecha, 527
 ciało
 metody, 73
 wyrażenia lambda, 351
 cykl życia serwletu, 1018
 czcionki, 762
 atrybuty, 768
 nazwy rodzaju, 762

nazwy rodziny, 762
 sprawdzanie dostępności, 763
 tworzenie, 765
 uzyskiwanie informacji, 767
 wybieranie, 765

D

datagramy, 708, 710
 definiowanie
 interfejsu, 203
 klasy, 128
 pakietu, 195
 własnych wyjątków, 229
 deklarowanie
 adnotacji, 273
 klasy, 128
 konstruktora kanonicznego, 415
 konstruktora niekanonicznego, 417
 metody domyślnej, 212
 modułu, 375
 obiektów, 131
 obiektu typu T, 318
 referencji do klasy sparametryzowanej, 323
 zmiennej typu wyliczeniowego, 260
 delegowanie
 interfejsu użytkownika, 935
 zdarzeń, 718, 733, 942
 deskryptory modułów, 376
 dodanie metody do klasy, 133
 domyślne implementacje metod interfejsów, 203
 dopasowywanie wzorców, 306, 420, 422, 909, 914
 dostawcy usług, 388
 dostęp
 do implementacji interfejsu, 204
 do kolekcji, 521
 do pakietów, 197
 do składowych, 173, 197
 dowiązanie
 późne, 191
 wczesne, 191
 drzewo, 969
 elementów, 972
 dynamiczne przydzielanie metod, 186
 działanie leniwe, 887
 dziedziczenie, inheritance, 49, 172, 191, 192
 hierarchia wielopoziomowa, 180
 wielokrotne, 214
 zapobieganie, 192

E

edytor tekstu TextArea, 792
 egzektory, 842, 860
 egzemplarz klasy, 49
 eksport kwalifikowany, 383
 etykieta, 124, 773
 ekranowa, tooltip, 985

F

filtr, 626
 Blur, 838
 Contrast, 836
 Grayscale, 834
 Invert, 834
 Sharpen, 839
 filtry konwolucyjne, 836
 finalne zgłoszenie dalej, 232
 format
 GIF, 817
 PNG, 817
 URL, 702
 formatowanie, 590
 daty i czasu, 593, 926, 921, 923, 927
 liczb, 592
 łańcuchów i znaków, 592
 tekstu, 907
 framework
 AWT, 733, 746
 Collections, 316, 503
 Fork/Join, 869, 873, 883
 JavaFX, 733
 Swing, 733
 funkcje, 49
 mieszające, 517
 trygonometryczne, 483
 wykładnicze, 484
 zaokrążeń, 484

G

gniazdo, socket, 695
 Berkeley, 695
 protokołu TCP/IP, 699, 707
 graf modułu, 395
 grupa wątków, 238
 grupowanie klauzul case, 401
 grupowe rozgłaszanie zdarzeń, 719

H

harmonogram udostępniania, 38
 hermetyzacja, encapsulation, 48, 49
 hierarchia
 AWT, 749
 dziedziczenia, 50
 klas sparametryzowanych, 338
 klas wyjątków, 219
 histogram, 826
 HSB, Hue-Saturation-Brightness, 759

I

identyfikator, 60
 klasy, 54
 URL, 707
 implementacja
 interfejsu, 204, 205
 Runnable, 239
 BinaryFunc, 390
 importowanie
 pakietów, 201
 statyczne, 311, 313
 indeks argumentu, 600
 inicjalizacja
 tablicy, 78
 zmiennej, 72
 instrukcja, *Patrz* słowo kluczowe
 interfejs, 202
 Action, 993
 ActionListener, 731, 785, 809
 AdjustmentListener, 731
 AnnotatedElement, 279, 280
 API dat i czasu, 924
 Appendable, 498
 AutoCloseable, 298, 499, 627
 BeanInfo, 1010
 BinaryFunc, 389
 BinaryOperator<T>, 374
 BinFuncProvider, 390
 Callable, 862
 CharSequence, 497
 Cloneable, 478
 Closeable, 298, 627
 Collection, 505
 Collector, 899
 Comparable, 498
 ComponentListener, 731
 Concurrent API, 841, 842

interfejs

- Consumer<T>, 374
- ContainerListener, 731
- Customizer, 1011
- Deque, 511
- Enumeration, 555
- ExecutorService, 860
- Externalizable, 658
- FilenameFilter, 626
- Flushable, 627
- FocusListener, 731
- Function<T,R>, 374
- Future, 862
- HttpResponse, 713
- HttpServletRequest, 1027
- HttpServletResponse, 1027
- HttpSession, 1028
- ImageConsumer, 825
- ImageProducer, 822
- IntStack, 214
- ItemListener, 731, 809
- Iterable, 499
- Iterator, 522
- Java Beans API, 1011
- java.io.Serializable, 1011
- KeyListener, 731
- LayoutManager, 794
- List, 507
- ListIterator, 522
- Map, 529
- Map.Entry, 534
- MouseListener, 732, 989
- MouseMotionListener, 732
- MouseWheelListener, 732
- NavigableMap, 533
- NavigableSet, 509
- ObjectInput, 660
- ObjectOutput, 659
- Path, 671
- PlugInFilter.java, 832
- Predicate<T>, 374
- publiczny, 49
- Queue, 510
- RandomAccess, 529
- Readable, 499
- Runnable, 237, 239, 487
- Serializable, 658
- Servlet API, 1021
- ServletConfig, 1022
- ServletContext, 1023
- ServletRequest, 1023
- ServletResponse, 1024
- Set, 508
- SharedConstants, 210
- SortedMap, 532
- SortedSet, 509
- Splititerator, 525, 903, 904
- StackWalker.StackFrame, 496
- Stream, 886, 887
- Supplier<T>, 374
- System.Logger, 478
- TextListener, 732
- Thread.UncaughtExceptionHandler, 500
- TreeNode, 970
- UnaryOperator<T>, 374
- WindowFocusListener, 732
- WindowListener, 732

interfejsy, 202

- atrybutów plików, 675
- definiowanie, 203
- dziedziczenie, 211
- funkcyjne, 44, 284, 350, 351, 373
- implementacja, 204
- kolekcji, 504
- łagodne modyfikowanie, 214
- map, 529
- metody
 - domyślne, 211, 214
 - prywatne, 216
 - statyczne, 215
- nasłuchujące zdarzeń, 730, 741
- pakietu
 - jakarta.servlet, 1021
 - java.beans, 1011
 - java.lang, 451
 - java.net, 696
 - java.util, 503, 616
 - java.util.function, 617–619
 - javax.servlet.http, 1026
- sparametryzowane, 334, 336, 357
- stosowanie, 207
- strumieni, 885
- usługi, 389
- wejścia-wyjścia, 622
- zagnieżdżone, 206
- zapięczętowane, 425
- zmienne typu final, 209

interpreter kodu bajtowego, 37

introspekcja, 501, 1008

IPv4, Internet Protocol, version 4, 695, 696

IPv6, Internet Protocol, version 6, 696

iterator, 503, 521–524, 902

J

jasność, 759

Java

- aplety, 35
- bezpieczeństwo, 36
- dynamika, 41
- ewolucja, 41
- interpretowalność, 41
- kod bajtowy, 36
- neutralność architektury, 41
- niezawodność, 40
- obiektowość, 40
- prostota, 40
- rozproszenie, 41
- serwlety, 39
- wielowątkowość, 41
- wydajność, 41

Java ARchive, 397

Java Beans, 1007

- introspekcja, 1008
- komponenty, 1008
- komponent typu Bean, 1013
- metody, 1010
- model zdarzeń, 1009
- trwałość, 1011
- właściwości ograniczone, 1010
- właściwości proste, 1008
- wzorce projektowe, 1010
- wzorce właściwości, 1008

java.base, 381

javac, 379, 383

jądro konwolucji, 836

jednostka kompilacji, 53

język

C, 31

C#, 35

C++, 33

Java, 33

jlink, 396, 398

jmod, 398

JVM, Java Virtual Machine, 36

K

kanały, 667, 669

katalog, 396, 625

klasa, 49, 128

AbstractAction, 994

ActionEvent, 721

AdjustmentEvent, 721

ArrayDeque, 520

ArrayList, 513, 889

Arrays, 550

AtomicInteger, 868

AtomicLong, 868

BitSet, 570

Blur.java, 838

Boolean, 267, 467

BorderLayout, 796

BufferedInputStream, 638

BufferedOutputStream, 640

BufferedReader, 293, 295, 652, 888

BufferedWriter, 654

Button, 774

Byte, 456

ByteArrayInputStream, 635

ByteArrayOutputStream, 637

Calendar, 576

Canvas, 750

CardLayout, 800

Character, 267, 464

CharArrayReader, 650

CharArrayWriter, 651

Checkbox, 778

CheckboxGroup, 780

CheckboxMenuItem, 807

Choice, 782

Class, 275, 480

ClassLoader, 483

ClassValue, 497

Collections, 545

Color, 752, 758

Compiler, 487

Component, 749, 818

ComponentEvent, 722

Console, 656

Container, 749, 794

ContainerEvent, 722

Contrast.java, 835

Convolver.java, 836

Cookie, 1029

CountDownLatch, 848

CropImageFilter, 828

Currency, 587

CyclicBarrier, 850

DatagramPacket, 709

DatagramSocket, 708

DataInputStream, 645

DataOutputStream, 645

Date, 575

DateFormat, 921

klasa

DateTimeFormatter, 926
 Dictionary, 560
 Double, 452
 Enum, 264, 496
 EnumMap, 538
 EnumSet, 521, 522
 Error, 219
 EventSetDescriptor, 1013
 Exception, 219, 229, 230
 Exchanger, 852
 File, 622
 FileChannel, 670
 FileInputStream, 298, 632
 FileOutputStream, 298, 301, 633
 FileReader, 649
 Files, 672
 FileStore, 676
 FileSystem, 676
 FileSystems, 676
 FileWriter, 649
 Float, 452
 FlowLayout, 795
 FocusEvent, 723
 Font, 763
 FontMetrics, 768
 ForkJoinPool, 871, 880, 882
 ForkJoinTask, 873, 881, 882
 ForkJoinTask<V>, 870
 Formatter, 588, 597, 601
 Frame, 750
 Gen, 317, 322
 GenericServlet, 1024
 Graphics, 737
 Grayscale.java, 834
 GregorianCalendar, 579
 GridBagLayout, 803
 GridLayout, 799
 HashMap, 535
 HashSet, 517
 Hashtable, 560
 HttpClient, 712
 HttpRequest, 713
 HttpServlet, 1030
 HttpSession, 1035
 HttpURLConnection, 705
 IdentityHashMap, 538
 Image, 817
 ImageFilter, 827
 ImageFilterDemo.java, 830
 ImageIcon, 949

Inet4Address, 699
 Inet6Address, 699
 InetAddress, 697
 InheritableThreadLocal, 493
 InputEvent, 724
 InputStream, 630
 InputStreamReader, 293
 Integer, 273, 456
 Introspector, 1013
 Invert.java, 834
 ItemEvent, 724
 JButton, 953
 JCheckBox, 957
 JCheckBoxMenuItem, 986
 JComboBox, 967
 JLabel, 949
 JList, 964
 JMenu, 977
 JMenuBar, 977
 JMenuItem, 978
 JRadioButton, 958
 JRadioButtonMenuItem, 986
 JScrollPane, 963
 JTabbedPane, 960
 JTable, 972
 JTextField, 951
 JToggleButton, 955
 JToolBar, 991
 JTree, 969
 KeyEvent, 725
 Label, 773
 LinkedHashMap, 537
 LinkedHashSet, 518
 LinkedList, 516
 List, 784
 ListResourceBundle, 611
 LoadedImage.java, 832
 LocalDate, 924
 LocalDateTime, 925
 Locale, 582
 LocalTime, 925
 Long, 456
 Matcher, 908
 Math, 312, 483
 MemoryImageSource, 822, 824
 MenuBar, 807
 MenuItem, 807
 MethodDescriptor, 1013
 Modifier, 916
 Module, 494
 ModuleLayer, 495

MouseEvent, 726
 MouseWheelEvent, 727
 Number, 324, 452
 Object, 193, 478
 ObjectInputStream, 661
 ObjectOutputStream, 659
 Optional, 572
 OptionalDouble, 572
 OptionalInt, 572
 OptionalLong, 572
 OutputStream, 296, 631
 Package, 493
 Panel, 749
 Paths, 674
 Pattern, 908
 Phaser, 853
 PixelGrabber, 825
 PrintStream, 296, 643
 PrintWriter, 297, 655
 PriorityQueue, 519
 Process, 468
 ProcessBuilder, 473
 Properties, 563
 PropertyDescriptor, 1013
 PropertyResourceBundle, 611
 PushbackInputStream, 640
 PushbackReader, 654
 Random, 210, 583
 RandomAccessFile, 646
 Reader, 293, 647
 Record, 497
 RecursiveAction, 871, 875
 RecursiveTask<V>, 871, 878
 ResourceBundle, 611
 RGBImageFilter, 830, 834
 Runtime, 469
 Runtime.Version, 471
 RuntimeException, 219
 RuntimePermission, 495
 Scanner, 602, 604, 606, 610
 Scrollbar, 787
 SecurityManager, 495
 Semaphore, 843
 SequenceInputStream, 641
 ServiceLoader, 388
 ServletInputStream, 1024
 ServletOutputStream, 1025
 Sharpen.java, 839
 Short, 456
 SimpleDateFormat, 922
 SimpleTimeZone, 581
 Socket, 699
 Stack, 558
 StackTraceElement, 495
 StackWalker, 496
 Stats, 324
 StrictMath, 487
 String, 162, 429
 StringBuffer, 163, 445, 449
 StringBuilder, 163, 450
 StringTokenizer, 568, 569
 Synch, 246
 System, 475
 System.LoggerFinder, 478
 TComp, 543
 TextEvent, 728
 TextField, 790
 Thread, 237, 240, 256, 487, 488
 ThreadGroup, 487, 490
 ThreadLocal, 493
 Throwable, 221, 229, 231, 305, 495
 Timer, 585
 TimerTask, 585
 TimeZone, 580
 TreeMap, 536, 538
 TreeSet, 518, 538
 UnicastRemoteObject, 918
 URI, 707
 URLConnection, 703
 Vector, 555
 Void, 468
 Window, 750
 WindowEvent, 728
 Writer, 647
 klasy, 49, 128
 abstrakcyjne, 189, 205
 adapterów, 736, 740, 741
 anonimowe, 744
 AWT, 747, 748
 bazowe, 50, 172
 biblioteki Swing, 949
 do obsługi dat i czasu, 924
 frameworku Fork/Join, 870
 kolekcji, 512
 map, 535
 opakowań, 452
 pakietu
 jakarta.servlet, 1022
 java.awt.event, 720
 java.awt.image, 817
 java.beans, 1012
 java.lang, 451

klasy

- java.lang.reflect, 915
- java.net, 696
- java.util, 502, 615
- java.util.concurrent, 842
- javax.servlet.http, 1027
- pochodne, 50, 172
- sparametryzowane, 318
 - bazowe, 338
 - przesłanianie metod, 343
 - tworzenie, 323
 - z dwoma parametrami typu, 322
- strumieni, 629
 - bajtów, 291
 - znaków, 291, 292
- systemu menu, 976
- systemu NIO, 666
- wartościowe, value-based classes, 315, 452
- wejścia-wyjścia, 622
- wewnętrzne, 160, 742
- wewnętrzne anonimowe, 162
- wyjątków, 219
- zagnieżdżone, 160
 - niestatyczne, 160
 - statyczne, 160
- zapieczętowane, 399, 423
- zdarzeń, 719, 720
- znaków, 912

klauzula

- case, 401, 403
- case ze strzałką, 406
- catch, 218, 220, 222
- extends, 211, 274, 324, 329
- finally, 218, 227
- throw, 218, 225
- throws, 218, 226

klawiatura, 737

kod

- bajtowy, 36
- rdzenny, native code, 308

kolejka LIFO, 141

- kolekcje, 316, 503, 567, 899
 - współbieżne, 865

kolory, 752, 758

komentarze, 53

- dokumentujące, 54, 61
- jednowierszowe, 54
- wielowierszowe, 54

komparatory, 538

kompilacja, 53

kompilator z wyprzedzeniem, 37

kompilowanie aplikacji, 379

komponent przeciwny, 723

komponenty

- biblioteki Swing, 936
- Java Beans, 1008

komunikacja międzywątkowa, 236, 249

konstruktory, 132, 138

- kanoniczne, 413

klasy

- bazowej, 177
- Checkbox, 778
- Color, 758
- Exception, 230
- Formatter, 589
- Frame, 750
- JToolBar, 991
- List, 784
- Scanner, 602
- Scrollbar, 787
- ServerSocket, 708
- Socket, 699
- String, 429
- StringBuffer, 445
- TextArea, 792
- TextComponent, 790

kolejność wykonywania, 183

niekanoniczne, 417

przeciążone, 146

- wywoływanie, 313

rekordów, 414, 418

skrótowe rekordu, 415

sparametryzowane, 139, 333

kontekst graficzny, 754

kontenery, 288

- biblioteki Swing, 937

najwyższego poziomu, 937

kontrola dostępu, 154

kontroler, 935

kontrolki

- AWT, 771
- dodawanie, 772
- usuwanie, 772
- zdarzenia, 772

konwersja

- automatyczna typów, 74
- danych, 442
- formatów, 588
- liczb, 462

kopiowanie pliku, 684

kwantyfikatory, 911

L

lambda, *Patrz* wyrażenia lambda
 liczby pseudolosowe, 210
 listy, 784
 obsługa zdarzeń, 785
 kombinowane, 967, 969
 komponentów, 413
 rozwijane, 782
 obsługa zdarzeń, 783
 stałych, 401
 literały, 909
 tekstowe, 431
 litery wzorca, 927
 logiczne wyrażenie AND, 421

Ł

łańcuchy, 85, 429
 długość, 431
 konkatenacja, 432
 konwersja, 433
 literały tekstowe, 431
 łączenie, 163, 443
 modyfikowanie, 439
 polecenia akcji, 953
 porównywanie, 435
 przeszukiwanie, 438
 wyjątków, 231
 wyodrębnianie znaków, 434
 zmiana wielkości liter, 442

M

mapy, 504, 529
 płaskie, flat maps, 899
 maszyna wirtualna Javy, JVM, 36
 mechanizm odzyskiwania pamięci, 141
 menedżer układu graficznego
 BorderLayout, 796
 CardLayout, 800
 FlowLayout, 794
 GridBagLayout, 803
 GridLayout, 799
 menu, 807
 dodanie
 pól wyboru, 987
 przycisków opcji, 987
 etykiet ekranowych, 984
 kombinacji klawiszy, 983
 mnemonik, 983
 obrazów, 984

główne, 979
 pakietu Swing, 975
 Plik, 984
 podręczne, 988, 990
 metoda, 49, 133
 accept(), 891, 903
 acquire(), 844
 actionPerformed(), 775, 830, 953, 959, 982,
 994, 996
 add(), 777, 782, 809
 addActionListener(), 943
 addCookie(), 1034
 addItem(), 967
 addMatch(), 906
 addMouseListener(), 736, 745, 989
 addMouseMotionListener(), 736
 addSuppressed(), 229
 addTab(), 960
 anyMatch(), 906
 append(), 447
 apply(), 892
 arraycopy(), 477
 arrive(), 854
 arriveAndAwaitAdvance(), 854, 856, 859
 arriveAndDeregister(), 857
 average(), 324
 await(), 850, 851
 bar(), 253
 boolean contentEquals (), 444
 bulkRegister(), 859
 call(), 246, 862
 cancel(), 880, 881
 capacity(), 445
 change(), 1014
 charAt(), 163, 357, 434, 446
 clear(), 214
 Clone(), 194
 clone(), 478
 close(), 298, 300, 304, 627, 628
 codePointAt (), 467
 codePointAt(), 444, 450, 467
 codePointBefore(), 444, 467
 codePointCount (), 444, 450
 collect(), 899, 901
 commonPool(), 872, 875
 compareAndSet(), 843, 868
 compareTo(), 264
 compute(), 871, 883
 concat(), 440
 contains (), 444
 contentEquals (), 444

metoda

convert(), 865
 count(), 906
 createImage(), 818
 createLineBorder(), 948
 currentThread(), 237, 238
 currentTimeMillis(), 476
 decrementAndGet(), 843, 868
 delete(), 448
 deleteCharAt(), 448
 destroy(), 1018, 1022
 distinct(), 906
 doGet(), 1031
 doPost(), 1032
 doubleValue(), 268, 269, 324
 drawArc(), 755
 drawLine(), 754
 drawOval(), 755
 drawRect(), 754
 drawString(), 737, 751, 753
 endsWith(), 436
 ensureCapacity(), 446
 equals(), 163, 193, 264, 413, 435
 equalsIgnoreCase(), 435
 execute(), 880
 exists(), 688
 Files.newByteChannel(), 682
 fillArc(), 755
 fillInStackTrace(), 229
 fillOval(), 755
 fillPolygon(), 755
 fillRect(), 754
 filter(), 832, 887, 891, 898
 filterRGB(), 834
 finalize(), 194
 find(), 908
 first(), 801
 flatMap(), 899
 flatMapToDouble(), 899
 flatMapToInt(), 899
 flatMapToLong(), 899
 flip(), 683
 flush(), 640
 foo(), 253
 forceTermination(), 860
 fork(), 870, 878
 format(), 444, 926
 formatted(), 444
 func(), 357, 390
 get(), 252, 390, 394, 668, 848
 getActionCommand(), 785, 953, 959

getAddListenerMethod(), 1013
 getAlignment(), 773
 getAllFonts(), 764
 getAndSet(), 843, 868
 getAnnotation(), 279, 289
 getAnnotations(), 278, 279
 getAnnotationsByType(), 289
 getAttribute(), 1035
 getAttributeNames(), 1035
 getAvailableFontFamilyNames(), 763, 764
 getBlue(), 759
 getBytes(), 434
 getCause(), 229, 231
 getChars(), 434, 447
 getClass(), 193, 194, 275
 getColor(), 760
 getComponent(), 989
 getContentPane(), 940
 getDateInstance(), 921
 getDateTimeInstance(), 922
 getEventSetDescriptors(), 1016
 getFirst(), 516
 getFont(), 767
 getGraphics(), 754, 820
 getGreen(), 759
 getHeight(), 769, 946
 getInsets(), 797
 getInsets(), 757, 945
 getItem(), 785, 809, 955, 957
 getItemCount(), 782, 785
 getItemSelectable(), 785
 getLabel(), 774
 getListenerType(), 1013
 getLocalGraphicsEnvironment(), 764
 getLocalizedMessage(), 229
 getMessage(), 229
 getMinimumSize(), 794, 832
 getMonth(), 928
 getName(), 238, 688, 1013, 1034
 getOb(), 319
 getParameter(), 1031
 getParent(), 688
 getPath(), 970
 getPreferredSize(), 794, 832
 getPriority(), 245
 getPropertyDescriptors(), 1010, 1014
 getQueuedTaskCount(), 882
 getRed(), 759
 getRGB(), 760
 getScrollType(), 728
 getSelectedCheckbox(), 780

getSelectedIndex(), 782, 785, 965
 getSelectedItem(), 782, 785
 getSelectedText(), 793
 getSelectedValue(), 966
 getSession(), 1035
 getSize(), 750, 757
 getSource(), 720, 776, 959
 getStackTrace(), 229
 getState(), 256, 257, 778, 808
 getSubElements(), 977
 getSuppressed(), 229, 305
 getSurplusQueuedTaskCount(), 882
 getText(), 773, 793, 951, 957
 getValue(), 352, 353, 787, 994, 1034
 getWidth(), 946
 getX(), 989
 grabPixel(), 826
 hashCode(), 194, 274, 413
 hasNext(), 604, 902
 HSBtoRGB(), 759
 indent(), 444
 indexOf(), 438, 450
 inForkJoinPool(), 881
 init(), 1018, 1022
 initCause(), 229, 231
 insert(), 447
 intValue(), 269
 invoke(), 872, 880
 invokeAll(), 875
 isAlive(), 243
 isBmpCodePoint(), 467
 isBound(), 700, 1013
 isCancelled(), 881
 isClosed(), 700
 isCompletedAbnormally(), 881
 isEditable(), 793
 isEmpty(), 444
 isExecutable(), 688
 isHidden(), 688
 isHighSurrogate(), 467
 isInfinite(), 455
 isNaN(), 455
 isPresent(), 890
 isReadable(), 688
 isSameAvg(), 325, 326
 isSelected(), 956–959, 977
 isSupplementaryCodePoint(), 467
 isSurrogatePair(), 467
 isValidCodePoint(), 467
 isWritable(), 688
 itemStateChanged(), 779, 783, 957
 iterator(), 902
 join(), 443, 870, 878
 keyPressed(), 737
 keyTyped(), 738
 lastIndexOf(), 438, 450
 length(), 163, 431, 445
 lines(), 444, 888
 list(), 625, 626
 listFiles(), 627
 load(), 388, 566
 lock(), 843
 m(), 270
 main(), 54, 702, 856
 map(), 681, 684, 898
 mapMulti(), 899
 mapMultiToInt(), 899
 mapMultiToLong(), 899
 mapToInt(), 898
 mark(), 636
 matcher(), 908
 matches(), 444, 910
 Math.pow(), 311
 Math.sqrt(), 311
 max(), 891
 min(), 890, 891
 mkdir(), 627
 mkdirs(), 627
 mouseClicked(), 741
 mouseDragged(), 741
 mouseMoved(), 822
 mousePressed(), 743, 989, 990, 1014
 mouseReleased(), 989, 990
 multclamp(), 835
 myMeth(), 274, 277
 myresume(), 255
 mysuspend(), 255
 newBuilder(), 713
 newByteChannel(), 681, 682
 newDirectoryStream(), 689
 newFileSystem(), 676
 newFixedThreadPool(), 860
 newHttpClient(), 713
 newInputStream(), 685, 686
 newOutputStream(), 685
 next(), 801, 902
 nextDouble(), 210
 noneMatch(), 906
 notify(), 193, 194, 249, 251, 841
 notifyAll(), 193, 194, 249
 now(), 925
 of(), 865, 906

metoda

offsetByCodePoints (), 444, 450
 ofInputStream(), 714
 onAdvance(), 857, 859
 ordinal(), 264
 paint(), 734, 751–754, 815, 822, 945
 paintComponent(), 945, 948
 parallel(), 893
 parallelStream(), 888, 893
 parse(), 928
 parseByte(), 462
 parseInt(), 462
 parseLong(), 462
 parseShort(), 462
 peekFirst(), 516
 pollFirst(), 516
 Polygon(), 755
 pop(), 209
 previous(), 801
 print(), 296, 297
 printf(), 602, 644
 println(), 194, 221, 238, 296, 643
 printStackTrace(), 229, 230
 push(), 209
 put(), 668, 669, 846, 848
 putValue(), 994
 read(), 294, 670, 818
 reader(), 294
 readLine(), 295
 readPassword(), 656
 reduce(), 891–895
 reinitialize(), 881
 release(), 844
 remove(), 940, 978
 removeActionListener(), 943
 removeAttribute(), 1035
 removeFirst(), 516
 removeKeyListener(), 719
 repaint(), 734, 741, 742, 752, 753
 replace(), 440, 449
 replaceAll(), 913
 replaceFirst (), 444
 reset(), 639
 resume(), 254
 reverse(), 448
 run(), 239, 248, 846, 881
 select(), 782, 793
 sequential(), 895
 service(), 1018, 1022
 set(), 832
 setAccelerator(), 983
 setActionCommand(), 777
 setAttribute(), 1035
 setBackground(), 752
 setBorder(), 948
 setCharAt(), 446
 setColor(), 760
 setEditable(), 793
 setForeground(), 752
 setJMenuBar(), 977
 setLabel(), 774
 setLayout(), 774, 794, 941
 setLength(), 446
 setMaxAge(), 1035
 setMnemonic(), 983
 setPaintMode(), 761
 setPriority(), 245
 setSelectedCheckbox(), 780
 setSize(), 750
 setStackTrace(), 230
 setState(), 778, 808
 setText(), 773, 793
 setTitle(), 751
 setUnitIncrement(), 788
 setVisible(), 751, 941
 setXORModel(), 761
 show(), 202
 sin(), 312
 sleep(), 238, 857, 865
 sort(), 887
 sorted(), 890
 split(), 444, 913
 start(), 239
 startsWith(), 436
 static int toCodePoint(), 467
 stop(), 254
 store(), 566
 stream(), 888, 889
 String replaceAll (), 444
 StringBuffer appendCodePoint(), 450
 stringWidth(), 769
 strip(), 441
 submit(), 862
 subSequence(), 445, 450
 substring(), 439, 449
 suspend(), 254
 System.exit(), 751
 System.nanoTime(), 876
 System.out.println(), 382
 test(), 891
 Thread.join(), 865
 Thread.sleep(), 865

- timedJoin(), 865
- toAbsolutePath(), 688
- toArray(), 515
- toCharArray(), 435
- toChars (), 467
- toChronoUnit(), 865
- toList(), 899
- toLowerCase(), 442
- toSet(), 899
- toString(), 193, 230, 413, 720, 882, 970
- toUpperCase(), 442
- transform(), 445
- trim(), 441
- trimToSize(), 450
- tryAdvance(), 903, 904
- tryLock(), 843
- trySplit(), 904
- unlock(), 843, 866
- unread(), 640, 654
- update(), 752, 753
- valueChanged(), 966, 970
- valueOf(), 261, 268, 442
- values(), 261
- wait(), 193, 194, 249, 251, 841
- walkFileTree(), 692
- write(), 296, 301, 670, 683
- writeTo(), 638
- metody, 49, 133
 - abstrakcyjne, 189, 560
 - dodawanie do klasy, 133
 - domyślne interfejsów, 203, 211–214
 - dynamiczne przydzielanie, 186
 - finalne, 191, 193
 - instancyjne, 364
 - interfejsu
 - BaseStream, 885
 - BasicFileAttributes, 675
 - CharSequence, 498
 - Collection, 506
 - Deque, 512
 - DosFileAttributes, 676
 - FileVisitor, 693
 - HttpServletRequest, 1027
 - HttpServletResponse, 1028
 - HttpSession, 1028, 1029
 - Iterator, 522
 - List, 507
 - ListIterator, 522
 - Map, 530, 531
 - Map.Entry, 534
 - NavigableMap, 533, 534
 - NavigableSet, 510
 - ObjectInput, 660
 - ObjectOutput, 659
 - Path, 671, 672
 - PosixFileAttributes, 676
 - Queue, 511
 - Servlet, 1022
 - ServletConfig, 1023
 - ServletContext, 1023
 - ServletRequest, 1023, 1024
 - ServletResponse, 1024
 - SortedMap, 532
 - SortedSet, 509
 - Splititerator, 525
 - Stream, 886, 887
- klasy
 - BitSet, 570, 571
 - Boolean, 468
 - Buffer, 667, 668
 - Byte, 456
 - Calendar, 577, 578
 - Character, 465
 - Class, 480, 482
 - Color, 759
 - Console, 657
 - Cookie, 1029
 - Currency, 588
 - DatagramPacket, 710
 - DatagramSocket, 709
 - Date, 575, 576
 - Dictionary, 560
 - Double, 454
 - Enum, 497
 - EnumSet, 522
 - File, 624, 625
 - Files, 672–674
 - Float, 453, 454
 - Font, 763
 - FontMetrics, 769
 - ForkJoinTask, 882
 - Formatter, 589, 597
 - Hashtable, 561
 - HttpResponse.BodyHandles, 714
 - HttpServlet, 1030
 - URLConnection, 705, 706
 - InetAddress, 698
 - InputStream, 630, 631
 - Integer, 458–460
 - Long, 460–462
 - Math, 485
 - Modifier, 916

metody

Object, 194, 478
 ObjectInputStream, 661, 662
 ObjectOutputStream, 659, 660
 Optional, 573, 574
 OutputStream, 631
 Package, 494
 Process, 469
 ProcessBuilder, 473, 474
 Properties, 564
 Random, 584
 Reader, 648
 ResourceBundle, 612, 613
 Runtime, 470
 Scanner, 604
 Short, 457, 458
 Socket, 700
 Stack, 558
 StackTraceElement, 496
 String, 443
 StringBuffer, 449
 StringTokenizer, 569
 System, 475, 476
 Thread, 237, 488–490
 ThreadGroup, 490
 Throwable, 229
 Timer, 586
 TimerTask, 586
 TimeZone, 581
 URLConnection, 703, 704
 Vector, 556
 Writer, 648
 mostu, 345
 napisane w kodzie rdzennym, 308
 o zmiennej liczbie argumentów, 167
 pobierające, 419, 1008
 prywatne w interfejsach, 216
 przeciążanie, 144, 167
 przesłanianie, 184, 187, 815
 w klasach sparametryzowanych, 343
 przyjmowanie parametrów, 136
 publiczne, 204
 rekurencyjne, 152
 rozszerzenia, 212
 rysujące figury, 756
 sparametryzowane, 332
 statyczne, 363
 statyczne interfejsów, 203, 215
 synchronizacja, 246
 ustawiające, 1008

varargs, 165
 wytwórcze, 257, 697, 714
 zapobieganie przesłanianiu, 191
 zmiennargumentowe, 165
 związane z blokadami, 866
 zwracanie obiektów, 151
 zwracanie wartości, 135
 mnemoniki, 983, 984
 model, 935
 delegowania zdarzeń, 718, 733, 942
 kolorów
 HSB, 759
 RGB, 759
 wątków, 235
 model-delegacja, Model-Delegate, 935
 moduł userfuncs, 392
 moduły, 375
 automatyczne, 398
 cechy, 395
 eksportowanie, 383
 nienazwane, unnamed modules, 382
 otwarte, 395
 platformy, 381
 wstępne, incubator modules, 426
 wymagania przechodnie, 384
 modyfikator
 strictfp, 308
 transient, 306
 volatile, 306
 modyfikatory
 dostępu, 54, 154, 198
 klasy, 916
 monitor, 236, 245
 muteks, 245
 mysz, 734

N

narzędzie, *Patrz* program
 nasycenie, 759
 nawiasy okrągłe, 101
 niejawna
 dostępność, 385
 zależność, 385
 niejednoznaczności wywołań, 168
 NIO, New I/O, 666
 NIO.2, 43, 671

O

- obiekt, 49
 - BufferedReader, 294
 - InputStream, 293
 - OutputStream, 297
 - PrintWriter, 297
 - String, 163
- obiekty
 - deklarowanie, 131
 - jako parametry, 148
 - nasłuchujące zdarzeń, 718, 719
- obramowania, insets, 757, 797
- obrazy graficzne, 817
 - ładowanie, 818
 - tworzenie, 818
 - wczytywanie, 818
 - wyświetlanie, 818, 819
- obserwator obrazu, 819
- obsługa
 - dat i czasu, 924
 - łańcuchów, 429
 - sieci, 695
 - wejścia i wyjścia, 290
 - wyjątków, 218
 - zdarzeń, 717, 941
 - żądań GET, 1031
 - żądań POST, 1032
- odczytywanie
 - danych z konsoli, 293
 - i zapisywanie plików, 297
 - łańcuchów, 295
 - znaków, 294
 - adnotacji, 278
- odzwierciedlanie, 275
- ograniczenia
 - typów sparametryzowanych, 348
 - var, 84
- okna
 - dialogowe, 812
 - określanie koloru tekstu, 752
 - określanie tła, 752
 - ponowne wyświetlenie zawartości, 752
 - typu Frame, 750
 - ukrywanie, 751
 - ustawianie tytułu, 751
 - ustawianie wymiarów, 750
 - wyświetlanie, 751
 - wyświetlanie łańcuchów znaków, 751
 - zamykanie, 751
- OOP, object-oriented programming, 33
- opakowania, 267, 269
 - typów numerycznych, 268
 - typów prostych, 451
 - dla typu boolean, 267
 - dla typu char, 267
- opcje otwierania plików, 674
- operacja
 - pop, 141
 - push, 141
- operacje
 - akumulacji, 892
 - atomowe, 868
 - końcowe, 887
 - redukcji, 891
 - wejścia-wyjścia, 621, 677, 685
- operator
 - &, 422
 - &&, 422
 - ==, 436
 - AND, 421
 - dekrementacji, 89
 - diamentu, 344
 - iloczynu bitowego, 91
 - inkrementacji, 89
 - instanceof, 306, 341, 420
 - kropki, 129
 - lambda, 351
 - negacji bitowej, 91
 - new, 131, 132, 226
 - porównania, 641
 - przesunięcia
 - w lewo, 93
 - w prawo, 94
 - w prawo bez znaku, 95
 - przypisania, 100, 641
 - reszty z dzielenia, 87
 - strzałki, 351
 - sumy
 - bitowej, 92
 - bitowej modulo 2, 92
 - logicznej, 232
 - trójargumentowy, 100
- operatory
 - arytmetyczne, 86
 - arytmetyczne z przypisaniem, 88
 - bitowe, 90
 - logiczne, 91
 - z przypisaniem, 96
 - logiczne, 98

operatory

logiczne ze skracaniem, 99

priorytety, 101

relacji, 97

opis wyjątku, 221

P

pakiet, 154, 195

appfuncs.simplefuncs, 379

Concurrency Utilities, 883

jakarta.servlet.http, 1026

java.awt.event, 719

java.awt.image, 817, 840

java.beans, 1011

java.io, 290, 298, 621, 687

java.lang, 228, 275, 311, 382, 451

podpakiety, 500

java.lang.annotation, 500

java.lang.constant, 500

java.lang.instrument, 500

java.lang.invoke, 500

java.lang.management, 500

java.lang.module, 501

java.lang.ref, 501

java.lang.reflect, 501, 907

java.net, 695

java.net.http, 711, 716

java.rmi, 907

java.text, 907, 921

java.time, 924, 928

java.time.format, 926

java.util, 502, 568, 615

java.util.concurrent, 616, 842, 864

java.util.concurrent.atomic, 616, 843, 868

java.util.concurrent.locks, 616, 843, 865

java.util.function, 617

java.util.jar, 619

java.util.logging, 619

java.util.prefs, 619

java.util.random, 619

java.util.regex, 619, 907

java.util.spi, 620

java.util.stream, 620, 885, 899

java.util.zip, 620

javax.imageio, 818, 840

javax.servlet, 1021

javax.swing, 936

javax.swing.event, 942

javax.swing.tree, 969

Swing, 933, 938, 949

userfuncs.binaryfuncs, 389

pakiety, 154, 195

biblioteki Swing, 938

definiowanie, 195

import, 201

systemu NIO, 666

uzyskiwanie dostępu, 197

znajdowanie, 196

panel

podzielony na zakładki, 960

szklany, 937

treści, 937

wielowarstwowy, 937

parametr, 54, 136

o zmiennej liczbie argumentów, 167

typu, T, 318

pasek

menu, 807

narzędzi, 991, 993

przewijania, 787, 964

obsługa zdarzeń, 788

pętla

do-while, 111

for, 58, 113

odmiany, 115

wykorzystanie przecinka, 115

zagnieżdżanie, 121

zmienna sterująca, 114

for-each, 116, 120, 524

while, 110

zdarzeń z odpytywaniem, 235

piaskownica, sandbox, 37

PLAF, pluggable look and feel, 934

pliki

.class, 274, 380, 691

.html, 626

.java, 691

automatyczne zamykanie, 303

cookie, 707

definicji modułów, 392

JAR, 396

JAR modularne, 397, 398

kopiowanie, 684

odczytywanie, 297

zapisywanie, 297, 682

podklasy sparametryzowane, 340

podpakiety pakietu java.util, 616

podwójne buforowanie, 820

pole tekstowe, 790

obsługa zdarzeń, 791

pole wyboru, 778, 957

obsługa zdarzeń, 779

polimorfizm, 50
 port, 695
 potok, 887
 precyzja, 596
 priorytety wątków, 236, 245
 procedury pośredniczące, 919
 proces, 234
 program
 jlink, 38, 398
 jpackage, 38
 jmod, 398
 MenuDemo, 998
 programowanie
 obiektove, OOP, 33, 47
 równoległe, parallel programming, 43, 235, 869
 strukturalne, 33
 wielowątkowe, 234
 współbieżne, concurrency utilities, 841
 protokół
 HTTP, 696, 1031
 IP, 695
 IPv4, 696
 IPv6, 696
 TCP, 695
 TCP/IP, 699
 UDP, 696
 przechwytywanie zmiennych, 362
 przeciążanie
 konstruktorów, 146
 metod, 144, 167
 przekazywanie argumentów
 przez referencję, 150
 przez wartość, 150
 przekazywanie komunikatów, 236
 przenośność, 36
 przesłanianie metod, 184, 187, 343
 przycisk, 774
 przełącznika, toggle button, 955
 przyciski
 biblioteki Swing, 953
 obsługa zdarzeń, 775
 opcji, 780, 958, 960
 pula wspólna, 871
 punkt
 kodowy, code point, 466
 wstrzymania, 996
 pusta instrukcja, 111

R

referencja, 131, 150
 super, 180
 this, 140, 284
 referencje
 do interfejsu, 204
 do konstruktorów, 369
 do metod
 instancyjnych, 364
 statycznych, 363
 typy sparametryzowane, 367
 do obiektów, 132, 204
 do obiektów klasy pochodnej, 176
 refleksja, reflection, 259, 275, 501, 907, 914
 rekordy, 399, 412
 płytko niezmiennicze, 413
 rekurencja, 152
 RGB, Red-Green-Blue, 759
 RMI, Remote Method Invocation, 41, 658, 907, 918
 rozgłaszanie zdarzenia, 1009
 rozpakowywanie, 269
 rozszerzanie klasy Thread, 240
 rozszerzenia switch, 400
 rysowanie
 elips, kół i okręgów, 755
 łuków, 755
 odcinków, 754
 prostokątów, 754
 w bibliotece Swing, 944, 946
 wielokątów, 755
 rzutowanie, 342
 niezgodnych typów, 75

S

SAM, Single Abstract Method, 351
 sekwencja if-else-if, 105
 sekwencje specjalne, escape sequences, 411
 selektory, 670
 semafor, 236, 245, 843
 separatory, 61, 568
 serializacja, 658, 662
 serwer Tomcat, 1019
 serwlety, 39, 1017
 cykl życia, 1018
 klasy wyjątków, 1025
 odczytywanie parametrów, 1025
 tworzenie, 1018, 1020

- sesje, 1035
- sieć, 695
- skaner, 568
- skanowanie, 603
- składowe klasy, 49, 129
 - kategorie widoczności, 198
 - statyczne, 157
 - uzyskiwanie dostępu, 197
- skrót, 517
- skrót klawiaturowy
 - Ctrl+Break, 254
 - Ctrl+C, 254
- słowa kluczowe, 61, 62
 - związane z modułami, 375
 - związane z usługami, 388
- słowo kluczowe
 - assert, 309
 - break, 106, 107, 118, 122, 123
 - case, 401, 403
 - catch, 218, 220, 222
 - class, 54, 128
 - continue, 125
 - default, 212
 - exports, 379, 380, 383
 - extends, 211, 274, 324, 329
 - final, 158, 191, 232
 - finally, 218, 227
 - goto, 123
 - if, 57, 103, 105
 - zagnieżdżanie, 104
 - implements, 204
 - import, 201
 - instanceof, 306
 - interface, 202, 274
 - module, 379
 - native, 309
 - new, 131, 132
 - opens, 396
 - package, 195
 - private, 154, 198
 - protected, 154, 198
 - provides, 389
 - public, 54, 154, 198
 - record, 412
 - requires static, 396
 - requires, 379, 380
 - return, 126
 - static, 157, 160, 216, 311
 - strictfp, 308
 - super, 177, 180
 - switch, 106, 399, 400, 402, 404–407
 - zagnieżdżanie, 109
 - synchronized, 247
 - this, 140, 313
 - throw, 218, 225
 - throws, 218, 226
 - transient, 306
 - try, 218, 220
 - zagnieżdżanie, 223
 - try-with-resources, 298, 304, 627, 629, 701
 - uses, 388, 389
 - var, 84
 - void, 138
 - volatile, 306
 - with, 389
 - yield, 402, 403

sparmetryzowane

 - interfejsy, 334
 - interfejsy funkcyjne, 357
 - klasy, 322
 - konstruktory, 333
 - metody, 332
 - podklasy, 340

specyfikator

 - konwersji formatu, 590
 - minimalnej szerokości pola, 595
 - precyzji, 596

specyfikatory

 - %n i %%, 595
 - formatów, 590, 591, 927
 - formatów pisane wielką literą, 599

spliteratory, 503, 525, 903

stała

 - BYTES, 462
 - MAX_VALUE, 462
 - MIN_VALUE, 462
 - SIZE, 462
 - TYPE, 462

stałe

 - całkowitoliczbowe, 68
 - klasowe, 276
 - klasy
 - GridBagConstraints, 804
 - MouseEvent, 726
 - MouseWheelEvent, 727
 - WindowEvent, 729
 - Float, 453
 - Double, 453
 - AdjustmentEvent, 721
 - Character, 464
 - ComponentEvent, 722

- ItemEvent, 725
- KeyEvent, 725
- Locale, 582
- logiczne, 70
- łańcuchowe, 71
- tekstowe, 163
- własnego typu, 260
- wyliczenia ElementType, 283
- wyliczeniowe, 260
- zmiennoprzecinkowe, 69
- znakowe, 70
- stan, 887
- stany wątku, 257
- stos, 141, 495
 - wywołań, 220
- stosowanie
 - akcji, 993
 - asercji, 309
 - datagramów, 710
 - frameworku Fork/Join, 873, 883
 - indeksu argumentu, 600
 - klasy Color, 760
 - komparatora, 540
 - kontrolerek typu TextArea, 792
 - list, 784
 - modelu delegowania zdarzeń, 733
 - modułów, 389
 - obramowań, 797
 - pól wyboru, 778
 - przycisków, 774
 - spliteratorów, 903
 - strumieni, 888, 902
 - strumieni równoległych, 893
 - systemu NIO, 677
 - typu Iterator, 902
 - typu wyliczeniowego, 265
 - usług, 388, 389
- strategia
 - CLASS, 274
 - dziel i zwyciężaj, 873
 - RUNTIME, 274
 - SOURCE, 274
 - zachowywania adnotacji, 274
- strona, party, 854
- strumienie, 291, 621, 884, 902
 - bajtów, 291, 630
 - buforowane, 638
 - filtrowane, 638
 - odwzorowywanie, 895
 - operacje redukcji, 891

- predefiniowane, 293
- reaktywne, reactive streams, 843
- równoległe, 888, 893
- uzyskiwanie, 887
- wejścia-wyjścia, 885
- znaków, 291, 647
- strumień
 - System.in, 294
 - System.out, 297
- sufiksy formatów daty i czasu, 593, 594
- suwak, 787
- Swing, *Patrz* biblioteka Swing
- synchronizacja, 236, 245
 - metod, 246
- synchronizatory, 842
- system NIO, 666

Ś

- ścieżka CLASSPATH, 196

T

- tabela, 974
- tablice, 54, 159
 - deklaracja, 77, 79, 80
 - deklaracja alternatywna, 82
 - inicjalizacja, 78
 - jednowymiarowe, 77
 - wielowymiarowe, 79, 119
- TCP, Transmission Control Protocol, 695
- tokeny, tokens, 568, 603
- Tomcat, 1019
- torba siatek, 803
- tryb
 - rysowania, 761
 - wielomodułowy, multimodule mode, 384
- tworzenie
 - aplikacji graficznych, 733
 - katalogów, 627
 - kolekcji, 899
 - konstruktorów rekordów, 414
 - menu głównych, 979
 - menu podręcznych, 988
 - metod sparametryzowanych, 332
 - obiektu obrazu, 818
 - paska narzędzi, 991
 - referencji do konstruktorów, 369
 - serwletów, 1018, 1020
 - wątku, 238, 257
 - wielu wątków, 241

typ

- byte, 64
- docelowy, target type, 350
- double, 66
- float, 66
- int, 65
- logiczny, 68
- long, 65
- short, 65
- T, 318, 325
- wyliczeniowy, 264
- wyliczeniowy TimeUnit, 864
- znakowy, 66

typy

- automatyczne rozszerzanie, 76
- całkowitoliczbowe, 64
- generyczne, *Patrz* typy sparametryzowane
- logiczne
 - automatyczne opakowywanie, 272
- numeryczne
 - opakowania, 268
- ograniczone, 323
- proste
 - automatyczne opakowywanie, 267, 269
- referencyjne, 170, 320
- SAM, 351
- sparametryzowane, generics, 316, 320
 - bezpieczeństwo, 320
 - błędy niejednoznaczności, 347
 - ograniczenia, 348
 - wnioskowanie typów, 344, 345
- surowe, 336
- wnioskowanie, 83
- wyjatków, 219
- wyliczeniowe, 259
- zmiennoprzecinkowe, 65
- znakowe
 - automatyczne opakowywanie, 272

U

- UDP, User Datagram Protocol, 696
- URI, Uniform Resource Identifier, 707
- URL, Uniform Resource Locator, 702, 1017
- uruchamianie
 - aplikacji, 379
 - przeglądarki, 1021
 - serwera Tomcat, 1021
- usługa, 388
- whois, 700

W

- warstwy, 398
- wartość początkowa, 583
- wątek, 234
 - główny, 237
 - rozdzielający zdarzenia, 941
- wątki
 - falszywe budzenie, 249
 - priorytety, 236, 245
 - problem wyścigu, 247
 - tworzenie, 238, 241, 257
 - uruchamianie, 257
 - uzyskiwanie stanu, 256
 - wznawianie, 254
 - zakleszczenie, 252
 - zatrzymywanie, 254
 - zawieszanie, 254
- wejście-wyjście, 290
- węzeł, 970
- widok, viewport, 935, 963
 - kolekcji, 504
- wielokrotne
 - dziedziczenie, 214
 - przechwytywanie wyjątków, 232
- wielowątkowość, 234, 258
- wirtualne kody klawiszy, 725
- własny typ, 260
- właściwości
 - ograniczone, 1010
 - proste, 1008
 - środowiska, 477
- wnioskowanie typów, 344
 - zmiennych lokalnych, 83, 120, 170, 192, 345
- wskaźnik pliku, 646
- współbieżność, 841
- wycieki pamięci, memory leaks, 298
- wyjątek, 218
 - ArithmeticException, 220, 223, 231, 233
 - ArrayIndexOutOfBoundsException, 222, 233
 - AssertionError, 309
 - EmptyArrayException, 361
 - FileNotFoundException, 298, 301
 - HeadlessException, 773
 - IOException, 294, 298, 628, 689
 - NumberFormatException, 268
 - PatternSyntaxException, 909
 - SecurityException, 298

wyjątki
 finalne zgłoszenie dalej, 232
 hierarchia klas, 219
 łańcuch, 231
 nieprzechwycone, 219
 nieweryfikowane, 228
 operacji wejścia-wyjścia, 628
 try-with-resources, 232
 tworzenie, 229
 wbudowane, 228
 weryfikowane, 228, 229
 wielokrotne przechwytywanie, 232
 wyświetlanie opisu, 221
 wykradanie zadań, work-stealing, 872
 wyliczenia, 259, 264
 jako typy klasowe, 262
 wartość kolejności, 264
 wyrażenia lambda, 44, 350, 352
 ciało blokowe, 355
 ciało wyrażeniowe, 355
 jako argumenty, 358
 przechwytywanie zmiennych, 362
 wyjątki, 361
 wyrażenia regularne, 603, 907
 przegląd, 914
 składnia, 909
 wyrażenie
 switch, 402, 404–407
 try-with-resources, 232, 298, 303–305
 wyścig, 247
 wyświetlanie informacji na konsoli, 296
 wywołanie
 konstruktora klasy bazowej, 177
 przeciążonych konstruktorów, 313
 this(), 313
 wzorce
 projektowe, 1010
 projektowe dla zdarzeń, 1009
 właściwości, 1008
 wzorzec, 907
 typu, type pattern, 421

Z

zadania
 anulowanie, 880
 asynchroniczne, 880
 określanie statusu wykonania, 881
 ponowne uruchamianie, 881
 zagnieżdżone instrukcje try, 223

zakleszczenie, 252, 941
 zakładki, 962
 zamykanie strumieni, 628
 zapisywanie pliku, 682
 zapobieganie
 dziedziczeniu, 192
 przesłanianiu, 191
 zasady rozszerzania typu, 76
 zbiór
 separatorów, 609
 znaczników, 597
 znaków, 293
 zdalne wywoływanie metod, RMI, 41, 658, 907, 918
 zdarzenia, 717, 718, 941, 1009
 klawiatury, 737
 kontrolki, 772
 list rozwijanych, 783
 listy, 785
 myszy, 734
 pasków przewijania, 788
 pól tekstowych, 791
 pól wyboru, 779
 przycisków, 775
 rozgłaszanie, 1009
 zdarzenie
 CaretEvent, 951
 TreeExpansionEvent, 969
 TreeModelEvent, 969
 zestawy znaków, 670
 zmienna
 liczba argumentów, 165, 168
 redukcja, mutable reduction, 902
 środowiskowa CLASSPATH, 196
 zmienne, 55, 71
 czas życia, 72
 deklaracja, 71
 inicjalizacja dynamiczna, 72
 iteracyjne, 116
 klasy bazowej, 176
 klasy Font, 763
 lokalne, 73
 praktycznie finalne, 303
 referencyjne, 132
 składowe, 49, 129
 ukrywanie, 140
 typu final, 209
 w interfejsach, 209
 wzorców, 421
 zasięg, 72

znacznik, 597

#, 599

czasowy zdarzenia, 721

nawiasów, 598

plusa, 598

przecinka, 599

spacji, 598

zera, 598

znaczniki

formatów, 597

kontekstu użytkownika, 1033

znak

\, 691

*, 691

?, 691

@, 274

cudzysłowu, 411

dwukropka, 363, 400, 404

gwiazdki, 201, 734

lewego ukośnika, 623

minusa, 597

odwrotnego ukośnika, 411

strzałki, 400, 351, 403

zapytania, 326

znaki

białe, 60

trzech kropek, 166

Unicode, 467

uzupełniające, supplemental characters, 466

zastępcze, 911

znoszenie, 345

zwracanie, pushback, 640

obiektów, 151

Ż

źródła zdarzeń, 718, 729

Ż

żądanie

GET, 1031

POST, 1032

PROGRAM PARTNERSKI

— GRUPY 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
Helion 

Java: wszystko, co musisz wiedzieć, w jednym kompendium!

Java od wielu lat utrzymuje swoją wyjątkową pozycję. To język programowania, który łączy w sobie technologiczną dojrzałość i nowoczesność. Jego twórcy od pierwszego wydania kładą nacisk na programowanie rozwiązań internetowych, a także na elastyczność i szybkość. W efekcie obecnie Java często jest pierwszym i najlepszym wyborem programistów, którzy chcą tworzyć szerokie spektrum oprogramowania. Wszędzie tam, gdzie wymaga się niezawodności, można znaleźć kod Javy.

Oto dwunaste wydanie wyczerpującego kompendium, w pełni zaktualizowane, uzupełnione o nowości wprowadzone w Java SE 17. Opisano w nim cały język Java: jego składnię, słowa kluczowe i najistotniejsze zasady programowania. Znalazły się tu także informacje dotyczące biblioteki Java API, takie jak operacje wejścia-wyjścia, biblioteka strumieni i techniki programowania współbieżnego. Nie zabrakło opisu biblioteki Swing, JavaBeans i serwletów, jak również licznych przykładów praktycznego zastosowania Javy. Wyczerpująco omówiono najnowsze możliwości języka, takie jak rekordy, klasy zapieczętowane czy też wyrażenia switch. Podręcznik został napisany w sposób przejrzysty, jasnym i zrozumiałym językiem, co znakomicie ułatwia naukę, a poszczególne zagadnienia zilustrowano licznymi przykładowymi fragmentami kodu źródłowego. To sprawia, że z kompendium skorzystają wszyscy programiści Javy, zarówno początkujący, jak i profesjonalni deweloperzy.

W książce między innymi:

- zasady programowania obiektowego
- obsługa zdarzeń, moduły i wyrażenia lambda
- klasy i interfejsy zapieczętowane
- rozszerzenia instrukcji switch, rekordy, bloki tekstu
- Collections, AWT, Swing, JavaBean i serwlety

Herbert Schildt jest niekwestionowanym autorytetem w dziedzinie programowania w Javie. Od ponad trzydziestu lat pisze książki poświęcone nauce Javy, C, C++ i C#; wiele z nich sprzedano w milionach egzemplarzy i przetłumaczono na różne języki. Pasjonuje się wszystkim, co jest związane z komputerowym przetwarzaniem, ale najbardziej językami programowania.

Helion 	KOD KORZYŚCI Sięgnij po więcej! ▶	
 helion.pl	ISBN 978-83-8322-156-4	
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	 9 788383 221564	
Cena: 199,00 zł		

