

Zgodne z JDK 7

Java



Przewodnik dla początkujących

Wydanie V

Chcesz nauczyć się Javy?
Zacznij już dziś!



Helion

Herbert Schildt



Tytuł oryginału: Java, A Beginner's Guide, 5th Edition

Tłumaczenie: Jaromir Senczyk

ISBN: 978-83-246-3919-9

Original edition copyright © 2012 by The McGraw-Hill Companies, Inc.
All rights reserved.

Polish edition copyright © 2012 by HELION SA
All rights reserved.

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

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

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

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

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/javpp5.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/javpp5>

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	11
O redaktorze technicznym	11
Wstęp	13
Rozdział 1. Podstawy Javy	19
Pochodzenie Javy	20
Java a języki C i C++	21
Java a C#	22
Java a Internet	22
Aplety Java	22
Bezpieczeństwo	23
Przeñośność	23
Magiczny kod bajtowy	24
Terminologia Javy	25
Programowanie obiektowe	26
Hermetyzacja	27
Polimorfizm	28
Dziedziczenie	28
Java Development Kit	29
Pierwszy prosty program	30
Wprowadzenie tekstu programu	30
Kompilowanie programu	31
Pierwszy program wiersz po wierszu	31
Obsługa błędów składni	34
Drugi prosty program	35
Inne typy danych	37
Przykład 1.1. Zamiana galonów na litry	38
Dwie instrukcje sterujące	39
Instrukcja if	40
Pęćła for	41
Bloki kodu	43
Średnik i pozycja kodu w wierszu	44
Wcięćcia	45
Przykład 1.2. Ulepszony konwerter galonów na litry	45
Słowa kluczowe języka Java	46
Identyfikatory	47
Biblioteki klas	48
Test sprawdzający	48

Rozdział 2. Typy danych i operatory	49
Dlaczego typy danych są tak ważne	50
Typy podstawowe	50
Typy całkowite	51
Typy zmiennoprzecinkowe	52
Znaki	53
Typ logiczny	54
Przykład 2.1. Jak daleko uderzył piorun?	55
Literały	56
Literały szesnastkowe, ósemkowe i binarne	57
Specjalne sekwencje znaków	57
Literały łańcuchowe	58
Zmienne	59
Inicjalizacja zmiennej	59
Dynamiczna inicjalizacja	60
Zasięg deklaracji i czas istnienia zmiennych	60
Operatory	63
Operatory arytmetyczne	63
Inkrementacja i dekrementacja	65
Operatory relacyjne i logiczne	66
Warunkowe operatory logiczne	67
Operator przypisania	69
Skrótowe operatory przypisania	69
Konwersje typów w instrukcjach przypisania	71
Rzutowanie typów niezgodnych	72
Priorytet operatorów	74
Przykład 2.2. Tabela prawdy dla operatorów logicznych	74
Wyrażenia	75
Konwersja typów w wyrażeniach	76
Odstępy i nawiasy	77
Test sprawdzający	78
Rozdział 3. Instrukcje sterujące	79
Wprowadzanie znaków z klawiatury	79
Instrukcja if	81
Zagnieżdżanie instrukcji if	82
Drabinka if-else-if	83
Instrukcja switch	84
Zagnieżdżanie instrukcji switch	88
Przykład 3.1. Rozpoczynamy budowę systemu pomocy	88
Pętla for	90
Wariacje na temat pętli for	92
Brakujące elementy	93
Pętla nieskończona	94
Pętle bez ciała	94
Deklaracja zmiennych sterujących wewnątrz pętli	95
Rozszerzona pętla for	96
Pętla while	96
Pętla do-while	97
Przykład 3.2. Ulepszamy system pomocy	99
Przerywanie pętli instrukcją break	102
Zastosowanie break jako formy goto	104
Zastosowanie instrukcji continue	108
Przykład 3.3. Końcowa wersja systemu pomocy	109

Pętle zagnieżdżone	112
Test sprawdzający	113
Rozdział 4. Wprowadzenie do klas, obiektów i metod	115
Podstawy klas	116
Ogólna postać klasy	116
Definiowanie klasy	117
Jak powstają obiekty	120
Referencje obiektów i operacje przypisania	120
Metody	121
Dodajemy metodę do klasy Vehicle	122
Powrót z metody	124
Zwracanie wartości	125
Stosowanie parametrów	127
Dodajemy sparametryzowaną metodę do klasy Vehicle	128
Przykład 4.1. System pomocy jako klasa	130
Konstruktory	135
Konstruktory z parametrami	136
Dodajemy konstruktor do klasy Vehicle	137
Operator new	138
Odzyskiwanie nieużytków i metoda finalize()	139
Metoda finalize()	139
Przykład 4.2. Ilustracja działania odzyskiwania nieużytków i metody finalize()	140
Słowo kluczowe this	142
Test sprawdzający	144
Rozdział 5. Więcej typów danych i operatorów	145
Tablice	145
Tablice jednowymiarowe	146
Przykład 5.1. Sortowanie tablicy	149
Tablice wielowymiarowe	151
Tablice dwuwymiarowe	151
Tablice nieregularne	152
Tablice o trzech i więcej wymiarach	153
Inicjalizacja tablic wielowymiarowych	153
Alternatywna składnia deklaracji tablic	155
Przypisywanie referencji tablic	155
Wykorzystanie składowej length	156
Przykład 5.2. Klasa Queue	158
Styl for-each pętli for	162
Iteracje w tablicach wielowymiarowych	165
Zastosowania rozszerzonej pętli for	166
Łańcuchy znaków	167
Tworzenie łańcuchów	167
Operacje na łańcuchach	168
Tablice łańcuchów	170
Łańcuchy są niezmiennicze	171
Łańcuchy sterujące instrukcją switch	172
Wykorzystanie argumentów wywołania programu	173
Operatory bitowe	175
Operatory bitowe AND, OR, XOR i NOT	175
Operatory przesunięcia	179
Skrótowe bitowe operatory przypisania	181
Przykład 5.3. Klasa ShowBits	182

Operator ?	184
Test sprawdzający	186
Rozdział 6. Więcej o metodach i klasach	189
Kontrola dostępu do składowych klasy	189
Modyfikatory dostępu w Javie	190
Przykład 6.1. Ulepszamy klasę Queue	194
Przekazywanie obiektów metodom	195
Sposób przekazywania argumentów	196
Zwracanie obiektów	199
Przeciążanie metod	201
Przeciążanie konstruktorów	205
Przykład 6.2. Przeciążamy konstruktor klasy Queue	207
Rekurencja	210
Słowo kluczowe static	212
Bloki static	215
Przykład 6.3. Algorytm Quicksort	216
Klasy zagnieżdżone i klasy wewnętrzne	218
Zmienne liczby argumentów	221
Metody o zmiennej liczbie argumentów	222
Przeciążanie metod o zmiennej liczbie argumentów	225
Zmienna liczba argumentów i niejednoznaczność	226
Test sprawdzający	227
Rozdział 7. Dziedziczenie	229
Podstawy dziedziczenia	230
Dostęp do składowych a dziedziczenie	232
Konstruktory i dziedziczenie	235
Użycie słowa kluczowego super do wywołania konstruktora klasy bazowej	237
Użycie słowa kluczowego super do dostępu do składowych klasy bazowej	240
Przykład 7.1. Tworzymy hierarchię klas Vehicle	241
Wielopoziomowe hierarchie klas	244
Kiedy wywoływane są konstruktory?	247
Referencje klasy bazowej i obiekty klasy pochodnej	248
Przesłanie metod	252
Przesłanie metod i polimorfizm	255
Po co przesłaniać metody?	257
Zastosowanie przesłaniania metod w klasie TwoDShape	257
Klasy abstrakcyjne	260
Słowo kluczowe final	264
final zapobiega przesłanianiu	264
final zapobiega dziedziczeniu	265
Użycie final dla zmiennych składowych	265
Klasa Object	267
Test sprawdzający	268
Rozdział 8. Pakiety i interfejsy	269
Pakiety	269
Definiowanie pakietu	270
Wyszukiwanie pakietów i zmienna CLASSPATH	271
Prosty przykład pakietu	272
Pakiety i dostęp do składowych	273
Przykład dostępu do pakietu	274
Składowe protected	275
Import pakietów	277

Biblioteka klas Java używa pakietów	279
Interfejsy	279
Implementacje interfejsów	281
Referencje interfejsu	284
Przykład 8.1. Tworzymy interfejs Queue	286
Zmienne w interfejsach	290
Interfejsy mogą dziedziczyć	291
Test sprawdzający	293
Rozdział 9. Obsługa wyjątków	295
Hierarchia wyjątków	296
Podstawy obsługi wyjątków	296
Słowa kluczowe try i catch	297
Prosty przykład wyjątku	298
Konsekwencje nieprzechwycenia wyjątku	300
Wyjątki umożliwiają obsługę błędów	301
Użycie wielu klauzul catch	302
Przechwytywanie wyjątków klas pochodnych	303
Zagnieżdżanie bloków try	304
Generowanie wyjątku	305
Powtórne generowanie wyjątku	306
Klasa Throwable	307
Klauzula finally	309
Użycie klauzuli throws	311
Nowości w JDK 7	312
Wyjątki wbudowane w Javę	313
Tworzenie klas pochodnych wyjątków	315
Przykład 9.1. Wprowadzamy wyjątki w klasie Queue	317
Test sprawdzający	320
Rozdział 10. Obsługa wejścia i wyjścia	323
Strumień wejścia i wyjścia	324
Strumień bajtowe i strumień znakowe	324
Klasy strumieni bajtowych	325
Klasy strumieni znakowych	326
Strumień predefiniowane	326
Używanie strumieni bajtowych	327
Odczyt wejścia konsoli	328
Zapis do wyjścia konsoli	329
Odczyt i zapis plików za pomocą strumieni bajtowych	330
Odczyt z pliku	330
Zapis w pliku	334
Automatyczne zamykanie pliku	336
Odczyt i zapis danych binarnych	339
Przykład 10.1. Narzędzie do porównywania plików	341
Pliki o dostępie swobodnym	343
Strumień znakowe	345
Odczyt konsoli za pomocą strumieni znakowych	345
Obsługa wyjścia konsoli za pomocą strumieni znakowych	349
Obsługa plików za pomocą strumieni znakowych	350
Klasa FileWriter	350
Klasa FileReader	351
Zastosowanie klas opakujących do konwersji łańcuchów numerycznych	352
Przykład 10.2. System pomocy wykorzystujący pliki	355
Test sprawdzający	361

Rozdział 11. Programowanie wielowątkowe	363
Podstawy wielowątkowości	364
Klasa Thread i interfejs Runnable	365
Tworzenie wątku	365
Drobne usprawnienia	369
Przykład 11.1. Tworzymy klasę pochodną klasy Thread	370
Tworzenie wielu wątków	372
Jak ustalić, kiedy wątek zakończył działanie?	375
Priorytety wątków	378
Synchronizacja	380
Synchronizacja metod	381
Synchronizacja instrukcji	384
Komunikacja międzywątkowa	386
Przykład użycia metod wait() i notify()	387
Wstrzymywanie, wznawianie i kończenie działania wątków	392
Przykład 11.2. Wykorzystanie głównego wątku	396
Test sprawdzający	397
Rozdział 12. Typy wyliczeniowe, automatyczne opakowywanie, import składowych statycznych i adnotacje	399
Wyliczenia	400
Podstawy wyliczeń	400
Wyliczenia są klasami	403
Metody values() i valueOf()	403
Konstruktory, metody, zmienne instancji a wyliczenia	404
Dwa ważne ograniczenia	406
Typy wyliczeniowe dziedziczą po klasie Enum	406
Przykład 12.1. Komputerowo sterowana sygnalizacja świetlna	408
Automatyczne opakowywanie	413
Typy opakowujące	413
Podstawy automatycznego opakowywania	415
Automatyczne opakowywanie i metody	416
Automatyczne opakowywanie i wyrażenia	418
Przeostroga	419
Import składowych statycznych	420
Adnotacje (metadane)	422
Test sprawdzający	425
Rozdział 13. Generyczność	427
Podstawy generyczności	428
Prosty przykład generyczności	428
Generyczność dotyczy tylko obiektów	432
Typy generyczne różnią się dla różnych argumentów	432
Klasa generyczna o dwóch parametrach	433
Ogólna postać klasy generycznej	434
Ograniczanie typów	434
Stosowanie argumentów wieloznacznych	438
Ograniczanie argumentów wieloznacznych	440
Metody generyczne	443
Konstruktory generyczne	445
Interfejsy generyczne	445
Przykład 13.1. Generyczna klasa Queue	448
Typy surowe i tradycyjny kod	452
Wnioskowanie typów i operator diamentowy	455
Wymazywanie	456

Błędy niejednoznaczności	457
Ograniczenia związane z generycznością	458
Zakaz tworzenia instancji parametru określającego typ	458
Ograniczenia dla składowych statycznych	458
Ograniczenia tablic generycznych	459
Ograniczenia związane z wyjątkami	460
Dalsze studiowanie zagadnienia generyczności	460
Test sprawdzający	461
Rozdział 14. Aplety, zdarzenia i pozostałe słowa kluczowe	463
Podstawy apletów	464
Organizacja apletów i podstawowe elementy	467
Architektura apletu	467
Kompletny szkielet apletu	468
Rozpoczęcie i zakończenie działania apletu	469
Żądanie odrysowania	470
Metoda update()	471
Przykład 14.1. Prosty aplet wyświetlający baner	471
Wykorzystanie okna statusu	475
Parametry apletów	475
Klasa Applet	477
Obsługa zdarzeń	479
Model delegacji zdarzeń	479
Zdarzenia	479
Źródła zdarzeń	479
Słuchacze zdarzeń	480
Klasy zdarzeń	480
Interfejsy słuchaczy zdarzeń	480
Wykorzystanie modelu delegacji zdarzeń	481
Obsługa zdarzeń myszy	482
Prosty aplet obsługujący zdarzenia myszy	483
Inne słowa kluczowe Javy	486
Modyfikatory transient i volatile	486
Operator instanceof	486
Słowo kluczowe strictfp	487
Słowo kluczowe assert	487
Metody natywne	488
Test sprawdzający	490
Rozdział 15. Wprowadzenie do Swing	491
Pochodzenie i filozofia Swing	492
Komponenty i kontenery	494
Komponenty	494
Kontenery	495
Panele kontenerów szczytowych	495
Menedżery układu	496
Pierwszy program wykorzystujący Swing	497
Pierwszy program Swing wiersz po wierszu	498
Komponent JButton	502
Komponent JTextField	506
Komponent JCheckBox	509
Komponent JList	512
Przykład 15.1. Porównywanie plików — aplikacja Swing	516
Wykorzystanie anonimowych klas wewnętrznych do obsługi zdarzeń	521
Aplety Swing	522

Co dalej?	524
Test sprawdzający	525
Dodatek A Rozwiązania testów sprawdzających	527
Rozdział 1. Podstawy Javy	527
Rozdział 2. Typy danych i operatory	529
Rozdział 3. Instrukcje sterujące	531
Rozdział 4. Wprowadzenie do klas, obiektów i metod	533
Rozdział 5. Więcej typów danych i operatorów	535
Rozdział 6. Więcej o metodach i klasach	538
Rozdział 7. Dziedziczenie	543
Rozdział 8. Pakiety i interfejsy	545
Rozdział 9. Obsługa wyjątków	546
Rozdział 10. Obsługa wejścia i wyjścia	549
Rozdział 11. Programowanie wielowątkowe	552
Rozdział 12. Typy wycieniowe, automatyczne opakowywanie, import składowych statycznych i adnotacje	554
Rozdział 13. Generyczność	558
Rozdział 14. Aplety, zdarzenia i pozostałe słowa kluczowe	562
Rozdział 15. Wprowadzenie do Swing	567
Dodatek B Komentarze dokumentacyjne	573
Znaczniki javadoc	573
@author	574
{@code}	575
@deprecated	575
{@docRoot}	575
@exception	575
{@inheritDoc}	575
{@link}	575
{@linkplain}	576
{@literal}	576
@param	576
@return	576
@see	576
@serial	577
@serialData	577
@serialField	577
@since	577
@throws	577
{@value}	578
@version	578
Ogólna postać komentarza dokumentacyjnego	578
Wynik działania programu javadoc	579
Przykład użycia komentarzy dokumentacyjnych	579
Skorowidz	581

Rozdział 11.

Programowanie wielowątkowe

W tym rozdziale poznasz:

- ◆ podstawy wielowątkowości,
- ◆ klasę Thread i interfejs Runnable,
- ◆ tworzenie wątku,
- ◆ tworzenie wielu wątków,
- ◆ sposób ustalania zakończenia wątku,
- ◆ korzystanie z priorytetów wątków,
- ◆ synchronizację wątków,
- ◆ synchronizację metod,
- ◆ synchronizację bloków,
- ◆ komunikację międzywątkową,
- ◆ wstrzymywanie, wznawianie i kończenie działania wątków.

Chociaż Java dostarcza programistom wielu innowacyjnych możliwości, to z pewnością jedną z najbardziej ekscytujących jest obsługa **programowania wielowątkowego**. Program wielowątkowy zawiera dwie lub więcej części, które mogą działać równolegle. Każdą z tych części nazywamy **wątkiem**, a każdy wątek definiuje osobną ścieżkę wykonania. Wielowątkowość jest zatem szczególną formą wielozadaniowości.

Podstawy wielowątkowości

Istnieją dwa podstawowe typy wielozadaniowości, z których jeden opiera się na procesach, a drugi wykorzystuje wątki. Ważne jest, abyś zrozumiał różnice między nimi. Proces odpowiada w zasadzie wykonywanemu programowi. Zatem wielozadaniowość *wykorzystująca procesy* umożliwia Twojemu komputerowi równoległe wykonywanie dwóch lub więcej programów. Wykorzystanie procesów umożliwi Ci na przykład uruchomienie kompilatora Javy w tym samym czasie, gdy piszesz kod źródłowy w edytorze lub przeglądasz strony w Internecie. W przypadku wielozadaniowości wykorzystującej procesy program jest najmniejszą jednostką kodu podlegającą szeregowaniu do wykonania przez procesor.

W drugim rodzaju wielozadaniowości najmniejszą jednostką kodu szeregowaną przez system jest wątek. Oznacza to, że pojedynczy program może równocześnie wykonywać dwa lub więcej wątków. Na przykład edytor tekstu może w tym samym czasie formatować jeden tekst i drukować inny, pod warunkiem że obie akcje są wykonywane przez osobne wątki. Chociaż Java wykorzystuje środowiska wielozadaniowe oparte na procesach, to nie umożliwia sterowania ich działaniem. W przypadku wątków sprawy mają się inaczej.

Podstawową zaletą wielowątkowości jest możliwość tworzenia efektywnie działających programów, ponieważ wielowątkowość pozwala wykorzystać okresy bezczynności pojawiające się w działaniu większości programów. Większość urządzeń wejścia i wyjścia takich jak porty sieciowe, napędy dyskowe czy klawiatura działa znacznie wolniej od procesora. Z tego powodu program często spędza wiele czasu na oczekiwaniu na możliwość wysłania lub odebrania danych za pośrednictwem urządzenia. Zastosowanie wielowątkowości pozwala programowi wykonywać w tym czasie inne zadania. Na przykład w czasie gdy jedna część programu wysyła plik w Internecie, inna może zajmować się odczytem klawiatury, a jeszcze inna buforować kolejny blok wysyłanych danych.

W ostatnich kilku latach mamy do czynienia z upowszechnieniem się systemów wieloprocesorowych i wielordzeniowych. Oczywiście nie wyparły one zupełnie systemów jednoprocessorowych. Wielowątkowość w Javie może działać w obu typach systemów. W przypadku systemu jednordzeniowego wykonywane równoległe wątki współdzielą procesor i każdy wątek otrzymuje pewien przedział czasu procesora. Zatem w systemie jednordzeniowym dwa lub więcej wątków nie jest w rzeczywistości wykonywanych równocześnie, ale ich zastosowanie pozwala wykorzystać czas bezczynności procesora. Natomiast w systemach wieloprocesorowych bądź wielordzeniowych wątki rzeczywiście mogą działać równoległe. W wielu przypadkach pozwala to jeszcze bardziej poprawić efektywność działania programu i zwiększyć szybkość wykonywania niektórych operacji.

Wątek może znajdować się w jednym z kilku stanów. Może być **wykonywany**. Może być **gotowy do wykonywania**, gdy tylko otrzyma czas procesora. Wykonanie wątku może zostać **zawieszony**, co oznacza, że wątek nie jest wykonywany przez pewien czas. Jego wykonywanie może zostać później **podjęte**. Wątek może zostać **zablokowany** w oczekiwaniu na zwolnienie pewnego zasobu. I wreszcie wykonywanie wątku może zostać **zakończone**, co oznacza, że nie można już go podjąć.

Wielowątkowość wiąże się nierozdzielnie z pojęciem **synchronizacji**, która umożliwia skoordynowane działanie wielu wątków. Java dysponuje kompletnym podsystemem synchronizacji. Jego podstawowe możliwości również omówię w tym rozdziale.

Jeśli programowałeś już na przykład w systemie Windows, to być może masz pewne pojęcie o wielowątkowości. Java umożliwia zarządzanie wątkami za pomocą odpowiednich elementów języka, co czyni programowanie wielowątkowe szczególnie wygodnym, ponieważ wiele jego szczegółów Java obsługuje za Ciebie.

Klasa Thread i interfejs Runnable

Wielowątkowość w Javie bazuje na klasie Thread i towarzyszącym jej interfejsie Runnable, które umieszczono w pakiecie `java.lang`. Klasa Thread hermetyzuje wątek. Jeśli chcesz stworzyć w programie nowy wątek, powinieneś utworzyć obiekt klasy pochodnej klasy Thread albo zaimplementować interfejs Runnable.

Klasa Thread definiuje szereg metod pomagających zarządzać wątkami. Najczęściej używane z tych metod przedstawiłem w tabeli 11.1 (przyjrzymy się im bliżej podczas ich użycia w przykładach).

Tabela 11.1. Wybrane metody klasy Thread

Metoda	Znaczenie
<code>final String getName()</code>	Zwraca nazwę wątku.
<code>final int getPriority()</code>	Zwraca priorytet wątku.
<code>final boolean isAlive()</code>	Sprawdza, czy wątek jest nadal wykonywany.
<code>final void join()</code>	Czeka na zakończenie wątku.
<code>void run()</code>	Punkt wejścia wątku.
<code>static void sleep(long milisekund)</code>	Zawieszanie wykonywanie wątku na podaną liczbę <i>milisekund</i> .
<code>void start()</code>	Rozpoczyna wykonywanie wątku przez wywołanie metody <code>run()</code> .

Wszystkie procesy mają przynajmniej jeden wątek wykonania, zwykle nazywany **wątkiem głównym**, ponieważ jego wykonanie rozpoczyna się w momencie uruchomienia programu. Możemy zatem powiedzieć, że we wszystkich dotychczasowych przykładach używaliśmy wątku głównego. W wątku głównym możesz tworzyć kolejne wątki programu.

Tworzenie wątku

Wątek powstaje przez utworzenie obiektu typu Thread. Klasa Thread hermetyzuje obiekt, który może być wykonywany. Jak już wspomniałem, Java umożliwia dwa sposoby tworzenia takich obiektów:

- ♦ poprzez implementację interfejsu Runnable,
- ♦ poprzez tworzenie klas pochodnych klasy Thread.

W większości przykładów w tym rozdziale będziemy implementować interfejs Runnable. Natomiast w przykładzie 11.1 zademonstruję, jak zaimplementować wątek, tworząc klasę

pochodną klasy `Thread`. Zapamiętaj: oba podejścia i tak używają klasy `Thread` do tworzenia wątku i zarządzania nim. Jedyna różnica polega na sposobie, w jaki powstaje klasa reprezentująca wątki.

Interfejs `Runnable` stanowi abstrakcję wykonywalnego kodu. Wątek możesz utworzyć na podstawie każdego obiektu, który implementuje interfejs `Runnable`. Interfejs ten ma tylko jedną metodę o nazwie `run()` zadeklarowaną w następujący sposób:

```
public void run()
```

Wewnątrz metody `run()` umieszczasz kod wykonywany przez nowy wątek. Podobnie jak główny wątek programu, tak i metoda `run()` może wywoływać inne metody, używać innych klas i deklarować zmienne. Jedyna różnica polega na tym, że metoda `run()` jest punktem wejścia do osobnego, równoległe wykonywanego wątku programu. Wykonywanie tego wątku kończy się, gdy metoda `run()` zwróci sterowanie.

Po utworzeniu klasy implementującej interfejs `Runnable` na podstawie jej obiektu tworzysz obiekt typu `Thread`. Klasa `Thread` definiuje szereg konstruktorów. Na początek będziemy używać poniższego:

```
Thread(Runnable obWątku)
```

W tym przypadku parametr `obWątku` jest instancją klasy implementującej interfejs `Runnable`. Definiuje on punkt, w którym rozpocznie się wykonywanie nowego wątku.

Po utworzeniu nowy wątek nie będzie wykonywany, dopóki nie wywołasz jego metody `start()` zadeklarowanej w klasie `Thread`. Działanie metody `start()` sprowadza się w zasadzie do wywołania metody `run()`. Poniżej przedstawiłem deklarację metody `start()`:

```
void start()
```

Na listingu 11.1 przedstawiłem przykład programu, który tworzy nowy wątek i rozpoczyna jego wykonywanie.

Listing 11.1. *UseThreads.java*

```
// Tworzy wątek, implementując interfejs Runnable.

class MyThread implements Runnable { ←
    String thrdName;

    MyThread(String name) {
        thrdName = name;
    }

    // Punkt wejściowy wątku.
    public void run() { ←          Tutaj rozpoczyna się wykonywanie wątku.
        System.out.println(thrdName + " rozpoczyna działanie.");
        try {
            for(int count=0; count < 10; count++) {
                Thread.sleep(400);
                System.out.println(thrdName +
                    " jest wykonywany, wartość licznika: " + count);
            }
        }
    }
}
```

Obiekty klasy `MyThread` mogą być wykonywane we własnych wątkach, ponieważ klasa `MyThread` implementuje interfejs `Runnable`.

```

        catch(InterruptedException exc) {
            System.out.println(thrdName + " został przerwany.");
        }
        System.out.println(thrdName + " kończy działanie.");
    }
}

class UseThreads {
    public static void main(String args[]) {
        System.out.println("Główny wątek rozpoczyna działanie.");

        // Najpierw tworzy obiekt klasy MyThread.
        MyThread mt = new MyThread("Wątek potomny nr 1"); ← Tworzy obiekt
                                                         implementujący
                                                         interfejs Runnable.

        // Następnie na jego podstawie tworzy wątek.
        Thread newThrd = new Thread(mt); ← Tworzy wątek dla tego obiektu.

        // Na koniec rozpoczyna wykonywanie wątku.
        newThrd.start(); ← Uruchamia wątek.

        for(int i=0; i<50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            }
            catch(InterruptedException exc) {
                System.out.println("Wątek główny został przerwany.");
            }
        }

        System.out.println("Wątek główny kończy działanie.");
    }
}

```

Przyjrzyjmy się bliżej temu programowi. Najpierw klasa `MyThread` implementuje interfejs `Runnable`. Oznacza to, że obiekt typu `MyThread` może zostać przekazany konstruktorowi klasy i być wykonywany we własnym wątku.

Wewnątrz metody `run()` działa pętla `for` odliczająca od 0 do 9. Zwróć uwagę na wywołanie metody `sleep()`. Metoda `sleep()` powoduje zawieszenie wątku, w którym została wywołana, na czas wyrażony w milisekundach. Jej deklarację przedstawiłem poniżej:

```
static void sleep(long milisekund) throws InterruptedException
```

Wykonywanie wątku zostaje zawieszona na czas *milisekund*. Metoda `sleep()` może wygenerować wyjątek `InterruptedException` i wobec tego musi być wywoływana w bloku `try`. Metoda `sleep()` ma również drugą wersję, która umożliwia określenie czasu zawieszenia wątku z dokładnością nie tylko co do milisekundy, ale również nanosekundy, jeśli potrzebujesz aż takiej precyzji. W naszym przykładzie metoda `sleep()` wywoływana przez metodę `run()` zawiesza wykonywanie wątku na 400 milisekund podczas każdego przebiegu pętli. Dzięki temu spowolnieniu możemy obserwować wykonywanie wątku.

Wewnątrz metody `main()` nowy obiekt typu `Thread` zostaje utworzony na skutek wykonania poniższej sekwencji instrukcji:

```
//Najpierw tworzy obiekt klasy MyThread.
MyThread mt = new MyThread("Wątek potomny nr 1");

//Następnie na jego podstawie tworzy wątek.
Thread newThrd = new Thread(mt);

//Na koniec rozpoczyna wykonywanie wątku.
newThrd.start();
```

Jak sugerują to komentarze, najpierw zostaje utworzony obiekt klasy `MyThread`. Obiekt ten zostaje następnie użyty do stworzenia obiektu typu `Thread`. Jest to możliwe, ponieważ klasa `MyThread` implementuje interfejs `Runnable`. Na koniec wykonywanie nowego wątku rozpoczyna się dzięki wywołaniu metody `start()`. Wywołuje ona metodę `run()` nowego wątku. Po wywołaniu metody `start()` sterowanie powraca do metody `main()`, która rozpoczyna wykonywanie własnej pętli `for`. Pętla ta wykonywana jest 50 razy i w każdym przebiegu zawieszona wykonanie głównego wątku na 100 milisekund. Oba wątki kontynuują swoje działanie, współdzieląc procesor (w systemie jednoprocessorowym). Trwa to aż do momentu, w którym zakończy się działanie wykonywanych przez nie pętli. Komunikaty wyświetlane przez oba wątki przedstawiłem poniżej. Ze względu na różnice w środowisku wykonania programu efekt jego działania w Twoim przypadku może się nieco różnić.

```
Główny wątek rozpoczyna działanie.
.Wątek potomny nr 1 rozpoczyna działanie.
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 0
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 1
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 2
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 3
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 4
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 5
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 6
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 7
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 8
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 9
Wątek potomny nr 1 kończy działanie.
.....Wątek główny kończy działanie.
```

W przykładzie tym warto zwrócić uwagę na jeszcze jedną rzecz. Aby zilustrować fakt równoczesnego wykonywania wątku głównego i wątku `mt`, musiałem unikać zakończenia działania metody `main()`, zanim wątek `mt` zakończy swoje działanie. W tym celu wykorzystałem różnice czasowe w działaniu obu wątków. Ponieważ wywołania metody `sleep()` w pętli `for` metody `main()` powodują całkowite opóźnienie równe 5 sekund (50 iteracji po 100 sekund każda), a całkowite opóźnienie wewnątrz pętli `for` metody `run()` wynosi jedynie 4 sekundy (10 iteracji po 400 milisekund każda), metoda `run()` zakończy działanie sekundę wcześniej niż metoda `main()`. W efekcie wątek główny i wątek `mt` są wykonywane równoległe do momentu zakończenia wątku `mt`. Wykonywanie wątku głównego (metody `main()`) kończy się sekundę później.

Chociaż wykorzystanie różnic czasowych pozwoliło mi zademonstrować równoległe działanie wątków, rozwiązania takiego nie stosuje się w praktyce. Java udostępnia znacznie lepsze sposoby oczekiwania na zakończenie wątku. Jeden z nich omówię w dalszej części rozdziału.

I jeszcze jedno: w przypadku programów wielowątkowych najczęściej będziemy chcieli, aby wątek główny zakończył swoje działanie jako ostatni. W ogólnym przypadku program działa,

dopóki nie zakończy się wykonywanie wszystkich jego wątków. Zatem nie jest wymagane, by wątek główny kończył swoje działanie jako ostatni. Ale często jest to przykładem dobrej praktyki programistycznej, zwłaszcza gdy dopiero uczysz się korzystania z wątków.

Drobne usprawnienia

Chociaż poprzedni program jest zupełnie poprawny, niewielkie modyfikacje mogą usprawnić jego działanie i ułatwić jego użycie. Po pierwsze, możemy rozpocząć wykonywanie wątku natychmiast po jego utworzeniu. W tym celu tworzymy obiekt `Thread` wewnątrz konstruktora klasy `MyThread`. Po drugie, klasa `MyThread` nie musi przechowywać nazwy wątku, ponieważ możemy nadać mu ją podczas tworzenia. W tym celu użyjemy następującej wersji konstruktora klasy `Thread`:

```
Thread(Runnable obWątku, String nazwa)
```

W tym przypadku parametr *nazwa* określa oczywiście nazwę wątku.

Ekspert odpowiada

Pytanie: Dlaczego zalecasz, aby główny wątek kończył działanie jako ostatni?

Odpowiedź: W starszych wersjach Javy zakończenie działania głównego wątku przed zakończeniem działania wątku potomnego mogło powodować błąd działania maszyny wirtualnej. Nowoczesne wersje Javy nie stwarzają już tego rodzaju problemów. Lepiej jednak zachować ostrożność, ponieważ nigdy nie wiemy, w jakim środowisku przyjdzie działać naszemu programowi. Dodatkowo wątek główny jest doskonałym miejscem do wykonania operacji związanych z zakończeniem działania programu, na przykład zamknięcia plików. Z tego powodu często rzeczywiście ma sens, aby kończył on swoje działanie jako ostatni. Na szczęście bardzo łatwo możemy zaprogramować oczekiwanie wątku głównego na zakończenie działania wszystkich jego wątków potomnych.

Nazwę wątku możemy uzyskać, wywołując metodę `getName()` zdefiniowaną w klasie `Thread`. Jej ogólną postać przedstawiłem poniżej:

```
final String getName()
```

Chociaż nie jest to konieczne, w naszym następnym programie możesz nadać nazwę wątkowi po jego utworzeniu. W tym celu użyjesz metody `setName()` przedstawionej poniżej:

```
final void setName(String nazwaWątku)
```

Ulepszoną wersję poprzedniego programu przedstawiłem na listingu 11.2.

Listing 11.2. *UseThreadsImproved.java*

```
// Ulepszona wersja klasy MyThread.
```

```
class MyThread implements Runnable {
    Thread thrd; ← Zmienna thrd przechowuje referencję wątku.

    // Tworzy nowy wątek.
    MyThread(String name) {
        thrd = new Thread(this, name); ← Wątek otrzymuje nazwę w momencie utworzenia.
        thrd.start(); // uruchamia nowy wątek ← Uruchamia wątek.
```

```

    }

    //Rozpoczyna wykonywanie nowego wątku.
    public void run() {
        System.out.println(thrd.getName() + " rozpoczyna działanie.");
        try {
            for(int count=0; count<10; count++) {
                Thread.sleep(400);
                System.out.println(thrd.getName() +
                    " jest wykonywany. wartość licznika: " + count);
            }
        } catch(InterruptedException exc) {
            System.out.println(thrd.getName() + " został przerwany.");
        }
        System.out.println(thrd.getName() + " kończy działanie.");
    }
}

class UseThreadsImproved {
    public static void main(String args[]) {
        System.out.println("Główny wątek rozpoczyna działanie.");

        MyThread mt = new MyThread("Wątek potomny nr 1");

        for(int i=0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            } catch(InterruptedException exc) {
                System.out.println("Wątek główny został przerwany.");
            }
        }

        System.out.println("Wątek główny kończy działanie.");
    }
}

```

↑ Teraz wątek zostaje uruchomiony w momencie utworzenia.

Ta wersja programu wyświetla takie same komunikaty jak poprzednia. Zwróć uwagę, że referencję wątku przechowuje składowa `thrd` klasy `MyThread`.

Przykład 11.1. Tworzymy klasę pochodną klasy `Thread`

ExtendThread.java

Implementacja interfejsu `Runnable` jest jednym ze sposobów tworzenia klasy reprezentującej wątki. Drugi sposób polega na utworzeniu klasy wątków jako klasy pochodnej klasy `Thread`. W tym przykładzie utworzymy właśnie klasę pochodną klasy `Thread` i wykorzystamy ją w programie, którego funkcjonalność będzie odpowiadać programowi `UseThreadsImproved`.

Tworząc klasę pochodną klasy `Thread`, musimy przesłonić jej metodę `run()` stanowiącą punkt wejścia do nowego wątku. Nowa klasa musi również wywołać metodę `start()`, aby rozpocząć wykonywanie nowego wątku. Możesz również przesłonić inne metody klasy `Thread()`, ale nie jest to wymagane.

1. Utwórz plik o nazwie *ExtendThread.java*. Skopiuj do niego kod źródłowy z pliku *UseThreadsImproved.java*.
2. Zmień deklarację klasy *Thread* tak, aby zamiast implementować interfejs *Runnable*, dziedziczyła po klasie *Thread*:

```
class MyThread extends Thread {
```

3. Usuń poniższy wiersz:

```
Thread thrd;
```

Zmienna składowa *thrd* nie jest już potrzebna, ponieważ instancja klasy *MyThread* zawiera instancję klasy *Thread*, do której może się odwoływać.

4. Zmień konstruktor klasy *MyThread* w następujący sposób:

```
// Tworzy nowy wątek.
MyThread(String name) {
    super(name); // nazwa wątku
    start(); // uruchamia wątek
}
```

Wywołanie *super* zostaje użyte w celu uruchomienia następującego konstruktora klasy *Thread*:

```
Thread(String nazwa)
```

gdzie parametr *nazwa* określa oczywiście nazwę tworzonego wątku.

5. Zmień metodę *run()* tak, aby bezpośrednio, czyli bez użycia zmiennej *thrd*, wywoływała metodę *getName()*:

```
// Rozpoczyna wykonywanie nowego wątku.
public void run() {
    System.out.println(getName() + " rozpoczyna działanie.");
    try {
        for(int count=0; count < 10; count++) {
            Thread.sleep(400);
            System.out.println(getName() +
                " jest wykonywany, wartość licznika: " + count);
        }
    }
    catch(InterruptedException exc) {
        System.out.println(getName() + " został przerwany.");
    }
    System.out.println(getName() + " został przerwany.");
}
```

6. Na listingu 11.3 przedstawiłem kompletny tekst źródłowy programu, który w obecnej wersji tworzy klasę pochodną klasy *Thread*, zamiast implementować interfejs *Runnable*. Wynik działania wyświetlany przez program będzie taki sam jak w przypadku poprzednich wersji programu.

Listing 11.3. *ExtendThread.java*

```
/*
```

Przykład 11.1

Tworzy klasę pochodną klasy Thread.

```

*/
class MyThread extends Thread {

    // Tworzy nowy wątek.
    MyThread(String name) {
        super(name); // nazwa wątku
        start(); // uruchamia wątek
    }

    // Rozpoczyna wykonywanie nowego wątku.
    public void run() {
        System.out.println(getName() + " rozpoczyna działanie.");
        try {
            for(int count=0; count < 10; count++) {
                Thread.sleep(400);
                System.out.println(getName() +
                    " jest wykonywany, wartość licznika: " + count);
            }
        }
        catch(InterruptedException exc) {
            System.out.println(getName() + " został przerwany.");
        }

        System.out.println(getName() + " został przerwany.");
    }
}

class ExtendThread {
    public static void main(String args[]) {
        System.out.println("Główny wątek rozpoczyna działanie.");

        MyThread mt = new MyThread("Wątek potomny nr 1");

        for(int i=0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            }
            catch(InterruptedException exc) {
                System.out.println("Wątek główny został przerwany.");
            }
        }

        System.out.println("Wątek główny kończy działanie.");
    }
}

```

Tworzenie wielu wątków

W poprzednich przykładach tworzyliśmy tylko jeden wątek potomny. Nic nie stoi jednak na przeszkodzie, aby Twój program tworzył dowolną, potrzebną liczbę wątków. Program przedstawiony na listingu 11.4 tworzy trzy wątki potomne.

Listing 11.4. *MoreThreads.java*

```

//Tworzy wiele wątków.

class MyThread implements Runnable {
    Thread thrd;

    //Tworzy nowy wątek.
    MyThread(String name) {
        thrd = new Thread(this, name);

        thrd.start(); //uruchamia wątek
    }

    //Rozpoczyna wykonywanie nowego wątku.
    public void run() {
        System.out.println(thrd.getName() + " rozpoczyna działanie.");
        try {
            for(int count=0; count < 10; count++) {
                Thread.sleep(400);
                System.out.println(thrd.getName() +
                    " jest wykonywany, wartość licznika: " + count);
            }
        }
        catch(InterruptedException exc) {
            System.out.println(thrd.getName() + " został przerwany.");
        }
        System.out.println(thrd.getName() + " kończy działanie.");
    }
}

class MoreThreads {
    public static void main(String args[]) {
        System.out.println("Główny wątek rozpoczyna działanie.");

        MyThread mt1 = new MyThread("Wątek potomny nr 1");
        MyThread mt2 = new MyThread("Wątek potomny nr 2"); ← Tworzy i uruchamia
        MyThread mt3 = new MyThread("Wątek potomny nr 3");      trzy wątki.

        for(int i=0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            }
            catch(InterruptedException exc) {
                System.out.println("Wątek główny został przerwany.");
            }
        }

        System.out.println("Wątek główny kończy działanie.");
    }
}

```

A oto przykład działania tego programu:

Główny wątek rozpoczyna działanie.
 Wątek potomny nr 1 rozpoczyna działanie.

```

Wątek potomny nr 3 rozpoczyna działanie.
.Wątek potomny nr 2 rozpoczyna działanie.
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 0
Wątek potomny nr 3 jest wykonywany, wartość licznika: 0
.Wątek potomny nr 2 jest wykonywany, wartość licznika: 0
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 1
Wątek potomny nr 3 jest wykonywany, wartość licznika: 1
.Wątek potomny nr 2 jest wykonywany, wartość licznika: 1
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 2
Wątek potomny nr 3 jest wykonywany, wartość licznika: 2
.Wątek potomny nr 2 jest wykonywany, wartość licznika: 2
...Wątek potomny nr 3 jest wykonywany, wartość licznika: 3
Wątek potomny nr 1 jest wykonywany, wartość licznika: 3
Wątek potomny nr 2 jest wykonywany, wartość licznika: 3
...Wątek potomny nr 3 jest wykonywany, wartość licznika: 4
Wątek potomny nr 1 jest wykonywany, wartość licznika: 4
Wątek potomny nr 2 jest wykonywany, wartość licznika: 4
...Wątek potomny nr 3 jest wykonywany, wartość licznika: 5
Wątek potomny nr 1 jest wykonywany, wartość licznika: 5
Wątek potomny nr 2 jest wykonywany, wartość licznika: 5
...Wątek potomny nr 3 jest wykonywany, wartość licznika: 6
Wątek potomny nr 1 jest wykonywany, wartość licznika: 6
.Wątek potomny nr 2 jest wykonywany, wartość licznika: 6
...Wątek potomny nr 3 jest wykonywany, wartość licznika: 7
Wątek potomny nr 1 jest wykonywany, wartość licznika: 7
.Wątek potomny nr 2 jest wykonywany, wartość licznika: 7
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 8
.Wątek potomny nr 3 jest wykonywany, wartość licznika: 8
Wątek potomny nr 2 jest wykonywany, wartość licznika: 8
...Wątek potomny nr 1 jest wykonywany, wartość licznika: 9
Wątek potomny nr 1 kończy działanie.
.Wątek potomny nr 3 jest wykonywany, wartość licznika: 9
Wątek potomny nr 3 kończy działanie.
Wątek potomny nr 2 jest wykonywany, wartość licznika: 9
Wątek potomny nr 2 kończy działanie.
.....Wątek główny kończy działanie.

```

Ekspert odpowiada

Pytanie: Dlaczego Java udostępnia dwa sposoby tworzenia wątków potomnych (przez tworzenie klas pochodnych klasy Thread lub przez implementację interfejsu Runnable) i który z nich jest lepszy?

Odpowiedź: Klasa Thread definiuje szereg metod, które mogą zostać przesłonięte w klasie pochodnej. Natomiast wymagane jest przesłonięcie tylko jednej z nich: run(). Ta sama metoda jest oczywiście wymagana podczas implementacji interfejsu Runnable. Część programistów Javy uważa, że klasy pochodne należy tworzyć tylko wtedy, gdy stanowią one specjalizację klasy bazowej. Jeśli zatem nie zamierzasz przesłaniać żadnej innej metody klasy Thread, to najlepiej implementuj interfejs Runnable. Implementacja interfejsu klasy Runnable umożliwi Twojej klasie wątku również dziedziczenie po innej klasie niż klasa Thread.

Jak widzisz, po uruchomieniu wszystkie trzy wątki potomne współdzielą czas procesora. Zwróć uwagę, że wątki są uruchamiane w kolejności ich utworzenia. Nie jest to regułą, Java może uszeregować działanie wątków w dowolny sposób. Ze względu na różnice w środowisku wykonywania informacje wyświetlane przez program mogą się nieco różnić w Twoim przypadku.

Jak ustalić, kiedy wątek zakończył działanie?

Często w programie chcielibyśmy sprawdzić, czy wątek zakończył swoje działanie. Na przykład w poprzednich przykładach w celach demonstracyjnych nie chciałem kończyć działania głównego wątku, zanim nie zakończyły się wątki potomne. W tym celu zawieszalem działanie wątku głównego na dłużej niż wątków potomnych. Jednak w ogólnym przypadku trudno uznać takie rozwiązanie za satysfakcjonujące.

Na szczęście klasa `Thread` dostarcza dwóch sposobów pozwalających ustalić, czy wątek zakończył działanie. Pierwszy z nich polega na wywołaniu metody `isAlive()` dla sprawdzanego wątku. Ogólną postać tej metody przedstawiłem poniżej:

```
final boolean isAlive()
```

Metoda `isAlive()` zwraca wartość `true`, jeśli wyjątek, dla którego została wywołana, nadal działa. W przeciwnym razie zwraca wartość `false`. Aby wypróbować jej działanie, zastąp w poprzednim przykładzie klasę `MoreThreads` jej wersją przedstawioną na listingu 11.5.

Listing 11.5. *MoreThreads2.java*

```
// Używa isAlive().
class MoreThreads2 {
    public static void main(String args[]) {
        System.out.println("Główny wątek rozpoczyna działanie.");

        MyThread mt1 = new MyThread("Wątek potomny nr 1");
        MyThread mt2 = new MyThread("Wątek potomny nr 2");
        MyThread mt3 = new MyThread("Wątek potomny nr 3");

        do {
            System.out.print(".");
            try {
                Thread.sleep(100);
            }
            catch (InterruptedException exc) {
                System.out.println("Wątek główny został przerwany.");
            }
        } while (mt1.thrd.isAlive() ||
                mt2.thrd.isAlive() || ← Oczekuje na zakończenie wszystkich wątków.
                mt3.thrd.isAlive());

        System.out.println("Wątek główny kończy działanie.");
    }
}
```

Ta wersja programu wyświetla podobne informacje jak poprzednia, z tą różnicą, że metoda `main()` kończy swoje działanie, jak tylko zakończą je pozostałe wątki. Różnica ta wynika z zastosowania metody `isAlive()` podczas oczekiwania na zakończenie wątków potomnych. Drugi sposób oczekiwania na zakończenie wątku polega na wywołaniu metody `join()` przedstawionej poniżej:

```
final void join( ) throws InterruptedException
```

Metoda ta czeka, aż wątek, dla którego została wywołana, zakończy działanie. Nazwa metody sugeruje, że wątek, który ją wywołał, czeka, aż określony wątek się z nim połączy (ang. *join*). Inne wersje metody `join()` pozwalają określić maksymalny czas oczekiwania na zakończenie wątku.

Na listingu 11.6 przedstawiłem program, który używa metody `join()`, aby zagwarantować, że główny wątek zakończy swoje działanie jako ostatni.

Listing 11.6. *JoinThreads.java*

```
// Używa join().

class MyThread implements Runnable {
    Thread thrd;

    // Tworzy nowy wątek.
    MyThread(String name) {
        thrd = new Thread(this, name);
        thrd.start(); // uruchamia nowy wątek
    }

    // Rozpoczyna wykonywanie nowego wątku.
    public void run() {
        System.out.println(thrd.getName() + " rozpoczyna działanie.");
        try {
            for(int count=0; count < 10; count++) {
                Thread.sleep(400);
                System.out.println(thrd.getName() +
                    " jest wykonywany, wartość licznika: " + count);
            }
        } catch(InterruptedException exc) {
            System.out.println(thrd.getName() + " został przerwany.");
        }
        System.out.println(thrd.getName() + " kończy działanie.");
    }
}

class JoinThreads {
    public static void main(String args[]) {
        System.out.println("Główny wątek rozpoczyna działanie.");

        MyThread mt1 = new MyThread("Wątek potomny nr 1");
        MyThread mt2 = new MyThread("Wątek potomny nr 2");
        MyThread mt3 = new MyThread("Wątek potomny nr 3");

        try {
            mt1.thrd.join(); ←
            System.out.println("Wątek potomny nr 1 zakończył działanie.");
            mt2.thrd.join(); ←
            System.out.println("Wątek potomny nr 2 zakończył działanie.");
            mt3.thrd.join(); ←
            System.out.println("Wątek potomny nr 3 zakończył działanie.");
        }
    }
}
```

Oczekują na zakończenie konkretnego wątku.


```

        catch(InterruptedException exc) {
            System.out.println("Wątek główny został przerwany.");
        }
        System.out.println("Wątek główny kończy działanie.");
    }
}

```

Poniżej przedstawiłem wynik działania tego programu. Pamiętaj, że gdy uruchomisz program w swoim środowisku, wynik może być nieco inny.

```

Główny wątek rozpoczyna działanie.
Wątek potomny nr 1 rozpoczyna działanie.
Wątek potomny nr 3 rozpoczyna działanie.
Wątek potomny nr 2 rozpoczyna działanie.
Wątek potomny nr 1 jest wykonywany, wartość licznika: 0
Wątek potomny nr 2 jest wykonywany, wartość licznika: 0
Wątek potomny nr 3 jest wykonywany, wartość licznika: 0
Wątek potomny nr 1 jest wykonywany, wartość licznika: 1
Wątek potomny nr 2 jest wykonywany, wartość licznika: 1
Wątek potomny nr 3 jest wykonywany, wartość licznika: 1
Wątek potomny nr 1 jest wykonywany, wartość licznika: 2
Wątek potomny nr 2 jest wykonywany, wartość licznika: 2
Wątek potomny nr 3 jest wykonywany, wartość licznika: 2
Wątek potomny nr 2 jest wykonywany, wartość licznika: 3
Wątek potomny nr 1 jest wykonywany, wartość licznika: 3
Wątek potomny nr 3 jest wykonywany, wartość licznika: 3
Wątek potomny nr 1 jest wykonywany, wartość licznika: 4
Wątek potomny nr 2 jest wykonywany, wartość licznika: 4
Wątek potomny nr 3 jest wykonywany, wartość licznika: 4
Wątek potomny nr 2 jest wykonywany, wartość licznika: 5
Wątek potomny nr 1 jest wykonywany, wartość licznika: 5
Wątek potomny nr 3 jest wykonywany, wartość licznika: 5
Wątek potomny nr 1 jest wykonywany, wartość licznika: 6
Wątek potomny nr 2 jest wykonywany, wartość licznika: 6
Wątek potomny nr 3 jest wykonywany, wartość licznika: 6
Wątek potomny nr 1 jest wykonywany, wartość licznika: 7
Wątek potomny nr 2 jest wykonywany, wartość licznika: 7
Wątek potomny nr 3 jest wykonywany, wartość licznika: 7
Wątek potomny nr 2 jest wykonywany, wartość licznika: 8
Wątek potomny nr 1 jest wykonywany, wartość licznika: 8
Wątek potomny nr 3 jest wykonywany, wartość licznika: 8
Wątek potomny nr 2 jest wykonywany, wartość licznika: 9
Wątek potomny nr 1 jest wykonywany, wartość licznika: 9
Wątek potomny nr 2 kończy działanie.
Wątek potomny nr 1 kończy działanie.
Wątek potomny nr 1 zakończył działanie.
Wątek potomny nr 2 zakończył działanie.
Wątek potomny nr 3 jest wykonywany, wartość licznika: 9
Wątek potomny nr 3 kończy działanie.
Wątek potomny nr 3 zakończył działanie.
Wątek główny kończy działanie.

```

Jak łatwo zauważyć, powrót sterowania z metody `join()` oznacza zakończenie wątku.

Priorytety wątków

Z każdym wątkiem związany jest priorytet. Priorytet wątku decyduje, przynajmniej częściowo, o tym, ile czasu procesora otrzyma dany wątek w porównaniu do innych wątków. W ogólnym przypadku spodziewamy się, że wątek o niskim priorytecie otrzyma mało czasu procesora, a wątek o wysokim priorytecie znacznie więcej. Ilość czasu procesora przydzielonego wątkowi ma istotny wpływ na charakterystykę wykonania wątku i jego interakcję z innymi wątkami wykonywanymi równolegle w systemie.

Na ilość czasu procesora przydzielaną wątkowi mają wpływ oprócz priorytetu wątku także inne czynniki. Na przykład: jeśli wątek o wysokim priorytecie oczekuje na pewien zasób, na przykład na wprowadzenie danych z klawiatury, to w tym czasie wykonywane będą wątki o niższym priorytecie. Jednak gdy wątek o wysokim priorytecie uzyska dostęp do zasobu, to wątki o niższym priorytecie zostaną wyłączone z procesora i wznowione zostanie wykonywanie wątku o wysokim priorytecie. Innym czynnikiem wpływającym na szeregowanie wątków jest sposób implementacji wielozadaniowości w danym systemie (patrz ramka „Ekspert odpowiada” na końcu tego podrozdziału). Dlatego też nawet jeśli nadasz różne priorytety wątkom programu, nie musi to oznaczać, że jeden z wątków będzie wykonywany szybciej lub częściej niż inny wątek. Wątek o wysokim priorytecie ma jedynie potencjalnie lepszy dostęp do procesora.

Wątek potomny ma po uruchomieniu taki sam priorytet jak wątek główny. Priorytet wątku możesz zmienić za pomocą metody `setPriority()` zdefiniowanej w klasie `Thread`. Poniżej przedstawiłem jej ogólną postać.

```
final void setPriority(int priorytet)
```

W tym przypadku parametr *priorytet* określa nową wartość priorytetu dla wątku, dla którego wywołano metodę. Wartość parametru *priorytet* musi należeć do przedziału od `MIN_PRIORITY` do `MAX_PRIORITY`. Obecnie stałe te mają wartości odpowiednio 1 i 10. Aby przywrócić wątkowi zwykły priorytet, podajesz wartość `NORM_PRIORITY`, obecnie równą 5. Wymienione wartości priorytetów zostały zadeklarowane jako `static final` w klasie `Thread`.

Bieżący priorytet wątku możesz uzyskać, wywołując metodę `getPriority()` zdefiniowaną w klasie `Thread`, którą przedstawiam poniżej:

```
final int getPriority( )
```

Przykład programu zamieszczony na listingu 11.7 demonstruje użycie dwóch wątków o różnych priorytetach. Wątki te tworzone są jako instancje klasy `Priority`. Metoda `run()` tej klasy zawiera pętlę zliczającą iteracje. Pętla kończy działanie, gdy wykona 10 000 000 iteracji lub gdy zmienna `stop` zadeklarowana jako `static` ma wartość `true`. Początkowo zmienna ta ma wartość `false`, a wartość `true` nadaje jej ten wątek, który pierwszy zakończy swoje działanie. W ten sposób drugi z wątków również kończy swoje działanie, gdy tylko otrzyma czas procesora. W każdym przebiegu pętli łańcuch `currentName` jest porównywany z nazwą wykonywanego wątku. Jeśli są różne, oznacza to, że nastąpiło przełączenie wątków. Wtedy wyświetlona zostaje nazwa nowego wątku, która zostaje również przypisana zmiennej `currentName`. Wyświetlanie informacji na skutek przełączenia wątku umożliwia użytkownikowi obserwację dostępu obu wątków do procesora. Po zakończeniu działania obu wątków program wyświetla liczbę iteracji obu pętli.

Listing 11.7. *PriorityDemo.java*

```

// Demonstruje priorytety wątków.

class Priority implements Runnable {
    int count;
    Thread thrd;

    static boolean stop = false;
    static String currentName;

    /* Tworzy nowy wątek. Zwróć uwagę, że ten
       konstruktor nie uruchamia wątków. */
    Priority(String name) {
        thrd = new Thread(this, name);
        count = 0;
        currentName = name;
    }

    // Rozpoczyna wykonywanie nowego wątku.
    public void run() {
        System.out.println(thrd.getName() + " rozpoczyna działanie.");
        do {
            count++;

            if(currentName.compareTo(thrd.getName()) != 0) {
                currentName = thrd.getName();
                System.out.println(currentName + " jest wykonywany.");
            }

        } while(stop == false && count < 10000000); ← Wątek, który pierwszy
        stop = true;                                osiągnie 10 000 000,
                                                    zatrzymuje wszystkie wątki.

        System.out.println("\n" + thrd.getName() +
                           " kończy działanie.");
    }
}

class PriorityDemo {
    public static void main(String args[]) {
        Priority mt1 = new Priority("Wątek o wysokim priorytecie");
        Priority mt2 = new Priority("Wątek o niskim priorytecie");

        // określa priorytet wątków
        mt1.thrd.setPriority(Thread.NORM_PRIORITY+2); ← Wątek mt1 otrzymuje
        mt2.thrd.setPriority(Thread.NORM_PRIORITY-2); ← wyższy priorytet niż wątek mt2.

        // uruchamia wątki
        mt1.thrd.start();
        mt2.thrd.start();

        try {
            mt1.thrd.join();
            mt2.thrd.join();
        }
        catch (InterruptedException exc) {
            System.out.println("Główny wątek rozpoczyna działanie.");
        }
    }
}

```

```

System.out.println("\nWątek o wysokim priorytecie odliczył do " +
                    mt1.count);
System.out.println("Wątek o niskim priorytecie odliczył do " +
                    mt2.count);
    }
}

```

A oto przykład działania programu:

Wątek o wysokim priorytecie rozpoczyna działanie.
 Wątek o wysokim priorytecie jest wykonywany.
 Wątek o niskim priorytecie rozpoczyna działanie.
 Wątek o niskim priorytecie jest wykonywany.
 Wątek o wysokim priorytecie jest wykonywany.

Wątek o niskim priorytecie kończy działanie.

Wątek o wysokim priorytecie kończy działanie.

Wątek o wysokim priorytecie odliczył do 10000000
 Wątek o niskim priorytecie odliczył do 8183

W tym przypadku wątek o wysokim priorytecie zawłaszczył większość czasu procesora. Oczywiście dokładny wynik działania programu zależy od liczby procesorów w systemie, ich szybkości, systemu operacyjnego i liczby innych zadań wykonywanych równocześnie przez system.

Ekspert odpowiada

Pytanie: Czy sposób implementacji wielozadaniowości w konkretnym systemie operacyjnym ma wpływ na to, ile czasu procesora otrzymuje wątek?

Odpowiedź: Oprócz priorytetu wątku największy wpływ na wykonanie wątku ma właśnie sposób implementacji wielozadaniowości i szeregowania zadań w systemie operacyjnym. Niektóre systemy stosują wielozadaniowość z wywłaszczaniem, co gwarantuje, że każdy wątek otrzymuje czas procesora, przynajmniej okazjonalnie. W systemach, które nie używają wywłaszczania, wykonywany wątek musi sam zwolnić procesor, zanim będzie możliwe wykonywanie innego wątku. W takich systemach łatwo o sytuację, w której jeden wątek zawłaszczy procesor, uniemożliwiając wykonywanie innych wątków.

Synchronizacja

Gdy używasz w programie wielu wątków, może pojawić się konieczność skoordynowania działania niektórych z nich. Proces, który to umożliwia, nazywamy **synchronizacją**. Najczęstszym powodem synchronizacji jest sytuacja, gdy dwa lub więcej wątków wymaga dostępu do współdzielonego zasobu, który może być używany w danej chwili tylko przez jeden wątek. Na przykład gdy jeden wątek zapisuje dane w pliku, inny wątek nie może wykonywać zapisu w tym samym pliku w tym samym czasie. Innym powodem stosowania synchronizacji jest oczekiwanie wątku na zdarzenie, którego przyczyną jest inny wątek. W tym przypadku musi istnieć sposób zawieszenia wykonywania wątku do momentu, w którym nastąpi wspomniane zdarzenie. Wtedy wątek musi wznowić wykonywanie.

Kluczem do synchronizacji wątków w Javie jest koncepcja **monitora**. Monitor kontroluje dostęp do obiektu, używając **blokad**. Gdy dostęp do obiektu zostanie zablokowany przez jeden wątek, obiekt nie jest dostępny dla innych wątków. Dopiero gdy wątek przestanie używać obiektu, blokada zostaje zwolniona i obiekt jest dostępny dla innych wątków.

Z każdym obiektem Javy związany jest monitor. Monitory zostały wbudowane w język Java i dzięki temu Java umożliwia synchronizację dostępu do wszystkich obiektów. Odbyna się ona przez zastosowanie słowa kluczowego `synchronized` i kilku metod, którymi dysponują wszystkie obiekty. Ponieważ Javę od początku projektowano z myślą o synchronizacji, posługiwanie się tym mechanizmem jest dużo łatwiejsze, niż mogłoby się wydawać. W wielu programach synchronizacja dostępu do obiektów odbywa się prawie w transparentny sposób.

Istnieją dwa sposoby zastosowania synchronizacji w Twoim kodzie. Oba wymagają użycia słowa kluczowego `synchronized` i oba zostaną omówione w tym rozdziale.

Synchronizacja metod

Synchronizację dostępu do metody możesz zapewnić, używając w jej deklaracji słowa kluczowego `synchronized`. Wywołanie takiej metody powoduje, że monitor blokuje dostęp do obiektu. Gdy obiekt jest zablokowany, żaden inny wątek nie może wywołać tej metody ani żadnej innej metody tego obiektu zadeklarowanej w jego klasie jako `synchronized`. Gdy działanie metody kończy się, monitor odblokowuje obiekt, umożliwiając tym samym jego użycie przez kolejny wątek. W ten sposób synchronizacja dostępu do obiektu odbywa się praktycznie bez jakiegokolwiek dodatkowego wysiłku ze strony programisty.

Program przedstawiony na listingu 11.8 demonstruje działanie mechanizmu synchronizacji na przykładzie metody `sumArray()` sumującej elementy tablicy zawierającej liczby całkowite.

Listing 11.8. *Sync.java*

```
// Synchronizacja dostępu do metody.

class SumArray {
    private int sum;

    synchronized int sumArray(int nums[]) { ← Dostęp do metody sumArray()
        sum = 0; // zeruje sumę                jest synchronizowany.

        for(int i=0; i<nums.length; i++) {
            sum += nums[i];
            System.out.println(Thread.currentThread().getName() +
                " wyliczył sumę częściową równą " + sum);
            try {
                Thread.sleep(10); // umożliwia przełączenie wątków
            }
            catch(InterruptedException exc) {
                System.out.println("Wątek został przerwany.");
            }
        }
    }
    return sum;
}
```

```

    }
}

class MyThread implements Runnable {
    Thread thrd;
    static SumArray sa = new SumArray();
    int a[];
    int answer;

    // Tworzy nowy wątek.
    MyThread(String name, int nums[]) {
        thrd = new Thread(this, name);
        a = nums;
        thrd.start(); // uruchamia wątek
    }

    // Rozpoczyna wykonywanie nowego wątku.
    public void run() {
        int sum;

        System.out.println(thrd.getName() + " rozpoczyna działanie.");

        answer = sa.sumArray(a);
        System.out.println(thrd.getName() +
            " wyliczył sumę równą " + answer);

        System.out.println(thrd.getName() + " kończy działanie.");
    }
}

class Sync {
    public static void main(String args[]) {
        int a[] = {1, 2, 3, 4, 5};

        MyThread mt1 = new MyThread("Wątek potomny nr 1", a);
        MyThread mt2 = new MyThread("Wątek potomny nr 2", a);

        try {
            mt1.thrd.join();
            mt2.thrd.join();
        }
        catch (InterruptedException exc) {
            System.out.println("Wątek główny został przerwany.");
        }
    }
}

```

A oto przykład działania tego programu (kolejność komunikatów może być w Twoim przypadku nieco inna).

```

Wątek potomny nr 1 rozpoczyna działanie.
Wątek potomny nr 2 rozpoczyna działanie.
Wątek potomny nr 1 wyliczył sumę częściową równą 1
Wątek potomny nr 1 wyliczył sumę częściową równą 3
Wątek potomny nr 1 wyliczył sumę częściową równą 6
Wątek potomny nr 1 wyliczył sumę częściową równą 10

```

```

Wątek potomny nr 1 wyliczył sumę częściową równą 15
Wątek potomny nr 2 wyliczył sumę częściową równą 1
Wątek potomny nr 1 wyliczył sumę równą 15
Wątek potomny nr 1 kończy działanie.
Wątek potomny nr 2 wyliczył sumę częściową równą 3
Wątek potomny nr 2 wyliczył sumę częściową równą 6
Wątek potomny nr 2 wyliczył sumę częściową równą 10
Wątek potomny nr 2 wyliczył sumę częściową równą 15
Wątek potomny nr 2 wyliczył sumę równą 15
Wątek potomny nr 2 kończy działanie.

```

Przeanalizujmy szczegółowo działanie tego programu. Tworzy on trzy klasy. Pierwszą z nich jest `SumArray`. Zawiera ona metodę `sumArray()`, która oblicza sumę elementów tablicy przechowywanej wartości całkowite. Druga klasa, `MyThread`, używa obiektu typu `SumArray`. Referencję sa tego obiektu zadeklarowałem jako `static` i wobec tego istnieje tylko jedna jej kopia współdzielona przez wszystkie obiekty klasy `MyThread`. Ostatnia z klas, `Sync`, tworzy dwa wątki obliczające sumę zawartości tej samej tablicy.

Metoda `sumArray()` wywołuje metodę `sleep()`, aby umożliwić przełączenie wątków. Jednak przełączenie nie jest możliwe, gdyż metoda `sumArray()` została zadeklarowana jako `synchronized` i wobec tego nie może być używana równocześnie przez dwa lub więcej wątków. Zatem gdy drugi z wątków potomnych rozpoczyna działanie, nie może wywołać metody `sumArray()`, dopóki nie wykona jej pierwszy wątek. Synchronizacja dostępu do metody gwarantuje w tym wypadku poprawny wynik jej działania.

Aby w pełni zrozumieć efekt słowa kluczowego `synchronized`, spróbuj usunąć je z deklaracji metody `sumArray()`. Dostęp do tej metody przestanie podlegać synchronizacji i wiele wątków będzie mogło jej używać równocześnie. Problem polega na tym, że suma częściowa jest przechowywana w składowej `sum`, która jest modyfikowana przez każdy wątek wywołujący metodę `sumArray()` dla obiektu sa zadeklarowanego jako `static`. Jeśli zatem dwa wątki wywołają równocześnie metodę `sa.sumArray()`, wyniki jej działania będą niepoprawne, ponieważ składowa `sum` będzie przechowywać na przemian sumy częściowe obliczane przez każdy z wątków. Poniżej przedstawiłem przykład działania programu po usunięciu słowa kluczowego `synchronized` z deklaracji metody `sumArray()` (wynik działania w Twoim przypadku może się różnić).

```

Wątek potomny nr 1 rozpoczyna działanie.
Wątek potomny nr 2 rozpoczyna działanie.
Wątek potomny nr 1 wyliczył sumę częściową równą 1
Wątek potomny nr 2 wyliczył sumę częściową równą 1
Wątek potomny nr 1 wyliczył sumę częściową równą 3
Wątek potomny nr 1 wyliczył sumę częściową równą 8
Wątek potomny nr 2 wyliczył sumę częściową równą 8
Wątek potomny nr 2 wyliczył sumę częściową równą 11
Wątek potomny nr 1 wyliczył sumę częściową równą 15
Wątek potomny nr 1 wyliczył sumę częściową równą 20
Wątek potomny nr 2 wyliczył sumę częściową równą 24
Wątek potomny nr 2 wyliczył sumę częściową równą 29
Wątek potomny nr 1 wyliczył sumę równą 24
Wątek potomny nr 1 kończy działanie.
Wątek potomny nr 2 wyliczył sumę równą 29
Wątek potomny nr 2 kończy działanie.

```

Powyższe informacje pokazują, że oba wątki potomne wywołują równocześnie metodę `sa.sumArray()`, przypisując składowej `sum` niewłaściwe wartości. Zanim przejdziemy do następnego zagadnienia, podsumujmy najważniejsze fakty związane z synchronizacją metod:

- ◆ synchronizacja metody polega na umieszczeniu słowa kluczowego `synchronized` w deklaracji metody;
- ◆ synchronizacja dostępu do metody powoduje, że jej wywołanie dla dowolnego obiektu powoduje jego zablokowanie i żaden inny wątek nie może wywołać dla tego samego obiektu żadnej metody zadeklarowanej jako `synchronized`;
- ◆ inne wątki próbujące wywołać metodę zadeklarowaną jako `synchronized` dla obiektu, do którego dostęp został zablokowany przez monitor, przechodzą w stan oczekiwania do momentu, w którym obiekt zostanie odblokowany;
- ◆ gdy wątek kończy wykonywanie metody `synchronized`, dostęp do obiektu zostaje odblokowany.

Synchronizacja instrukcji

Chociaż deklarowanie metod jako `synchronized` stanowi prosty i efektywny sposób wykorzystania mechanizmu synchronizacji, nie sprawdza się on we wszystkich sytuacjach. Na przykład może wystąpić konieczność zapewnienia synchronizacji dostępu do metody, która nie została zadeklarowana jako `synchronized`. Może się to zdarzyć, gdy używasz cudzych klas i nie masz dostępu do ich kodu źródłowego. Nie możesz zatem dodać słowa kluczowego `synchronized` w deklaracji interesującej Cię metody. W jaki sposób możesz zatem zapewnić synchronizację dostępu do obiektu tej klasy? Rozwiązanie tego problemu jest na szczęście bardzo proste: wystarczy umieszczać wywołania metod tej klasy w bloku oznaczonym jako `synchronized`.

Poniżej przedstawiłem ogólną postać bloku `synchronized`:

```
synchronized(refobj) {
    // instrukcje wymagające synchronizacji
}
```

W tym przypadku `refobj` jest referencją obiektu, do którego dostęp wymaga synchronizacji. Gdy wątek rozpocznie wykonywanie bloku `synchronized`, żaden inny wątek nie może wywołać metody dla obiektu `refobj`, dopóki bieżący wątek nie opuści tego bloku.

Kolejny sposób synchronizacji wywołań metody `sumArray()` polega zatem na ich umieszczeniu w bloku `synchronized`. Ilustruje to wersja programu przedstawiona na listingu 11.9.

Listing 11.9. *Sync2.java*

```
// Używa bloku synchronized
// dla synchronizacji dostępu
// do metody sumArray.
class SumArray {
    private int sum;
```



```

int sumArray(int nums[]) { ←————— W tym wypadku dostęp do metody sumArray()
    sum = 0; //zeruje sumę                               nie podlega synchronizacji.

    for(int i=0; i<nums.length; i++) {
        sum += nums[i];
        System.out.println(Thread.currentThread().getName() +
            " wyliczył sumę częściową równą " + sum);
        try {
            Thread.sleep(10); //umożliwia przełączenie wątków
        }
        catch(InterruptedException exc) {
            System.out.println("Wątek został przerwany.");
        }
    }
    return sum;
}

class MyThread implements Runnable {
    Thread thrd;
    static SumArray sa = new SumArray();
    int a[];
    int answer;

    //Tworzy nowy wątek.
    MyThread(String name, int nums[]) {
        thrd = new Thread(this, name);
        a = nums;

        thrd.start(); //uruchamia wątek
    }

    //Rozpoczyna wykonywanie nowego wątku.
    public void run() {
        int sum;

        System.out.println(thrd.getName() + " rozpoczyna działanie.");

        //synchronizuje wywołania metody sumArray()
        synchronized(sa) { ←————— Synchronizacja wywołań metody sumArray() dla obiektu sa.
            answer = sa.sumArray(a);
        }
        System.out.println(thrd.getName() +
            " wyliczył sumę równą " + answer);

        System.out.println(thrd.getName() + " kończy działanie.");
    }
}

class Sync2 {
    public static void main(String args[]) {
        int a[] = {1, 2, 3, 4, 5};

        MyThread mt1 = new MyThread("Wątek potomny nr 1", a);
        MyThread mt2 = new MyThread("Wątek potomny nr 2", a);

        try {
            mt1.thrd.join();

```

```

        mt2.thrd.join();
    } catch (InterruptedException exc) {
        System.out.println("Wątek główny został przerwany.");
    }
}
}
}

```

Wynik działania tej wersji programu będzie poprawny, taki sam jak w przypadku wykorzystania metody zadeklarowanej jako `synchronized`.

Ekspert odpowiada

Pytanie: Słyszałem coś o „API współbieżności”. Do czego one służą? Do czego służy szkielet Fork/Join?

Odpowiedź: API współbieżności udostępniono w pakiecie `java.util.concurrent` (i jego pakietach podrzędnych) w celu umożliwienia w Javie programowania współbieżnego. Klasy te implementują między innymi synchronizatory, pule wątków, menedżery wykonania i blokady umożliwiające sterowanie wykonaniem wątków. Jedną z najciekawszych możliwości oferowanych przez API współbieżności jest szkielet Fork/Join wprowadzony w JDK 7.

Szkielet ten umożliwia tak zwane **programowanie równoległe**. Nazwy tej używa się do określenia technik pozwalających wykorzystać zalety komputerów o dwu lub więcej procesorach (włączając w to systemy wielordzeniowe) poprzez podział zadania na podzadania i wykonywanie każdego podzadania na osobnym procesorze. Rozwiązanie takie pozwala zwiększyć szybkość wykonywania działania i poprawić efektywność systemu. Podstawową zaletą szkieletu Fork/Join jest łatwość użycia. Upraszcza on tworzenie kodu wielowątkowego, który automatycznie skaluje się tak, aby wykorzystać liczbę procesorów dostępnych w systemie. Tym samym sprowadza tworzenie rozwiązań współbieżnych do poziomu nie trudniejszego niż typowe zadania programowania, takie jak na przykład operacje na elementach tablicy. API współbieżności, a w szczególności szkielet Fork/Join, warte są poznania, gdy nabierzesz większego doświadczenia w tworzeniu programów wielowątkowych.

Komunikacja międzywątkowa

Rozważmy następującą sytuację. Wątek T wykonuje właśnie metodę zadeklarowaną jako `synchronized` i wymaga dostępu do zasobu R, który jest tymczasowo niedostępny. Co powinien zrobić wątek T? Jeśli zacznie sprawdzać w pętli dostępność zasobu R, to zablokuje używany obiekt, uniemożliwiając dostęp innym wątkom. Rozwiązanie takie jest dalekie od optymalnego, ponieważ częściowo redukuje zalety programowania wielowątkowego. Lepiej będzie, jeśli wątek T zwolni tymczasowo wykorzystywany obiekt, umożliwiając wykonywanie innych wątków. Gdy zasób R stanie się dostępny, wątek T zostanie powiadomiony i wznowi działanie. Takie rozwiązanie wymaga jednak komunikacji międzywątkowej, dzięki której jeden wątek może powiadomić inne, że został zablokowany, a sam zostać powiadomiony, że może wznowić działanie. Java umożliwia komunikację międzywątkową za pomocą metod `wait()`, `notify()` i `notifyAll()`.

Metody `wait()`, `notify()` i `notifyAll()` są dostępne dla każdego obiektu, ponieważ zostały zaimplementowane w klasie `Object`. Metody te mogą być wywoływane tylko w kontekście

działania mechanizmu synchronizacji. Używamy ich w następujący sposób. Gdy wątek zostaje czasowo zablokowany, wywołuje metodę `wait()`. Wykonanie wątku zostaje zawieszona, a monitor obiektu zwalnia blokadę, umożliwiając innemu wątkowi dostęp do obiektu. Zawieszony wątek wznowi później swoje działanie, gdy inny wątek wywoła dla tego samego obiektu metodę `notify()` lub `notifyAll()`.

Poniżej przedstawiłem różne wersje metody `wait()` zdefiniowane w klasie `Object`:

```
final void wait() throws InterruptedException
final void wait(long milisekund) throws InterruptedException
final void wait(long milisekund, int nanosekund) throws InterruptedException
```

Pierwsza wersja zawiesza wykonywanie wątku do momentu wywołania metody `notify()` lub `notifyAll()`. Druga podobnie, ale wątek nie będzie oczekiwać dłużej niż podaną liczbę *milisekund*. Trzecia wersja umożliwia jeszcze dokładniejsze określenie maksymalnego czasu oczekiwania z dokładnością do pojedynczych *nanosekund*.

Poniżej przedstawiłem ogólną postać metod `notify()` i `notifyAll()`:

```
final void notify()
final void notifyAll()
```

Wywołanie metody `notify()` powoduje wznowienie wykonywania oczekującego wątku. Wywołanie metody `notifyAll()` powiadamia wszystkie wątki, a dostęp do obiektu uzyskuje wątek o najwyższym priorytecie.

Zanim przyjrzymy się przykładowi zastosowania tych metod, muszę uczynić jeszcze jedną ważną uwagę. Choć w normalnych warunkach metoda `wait()` powoduje oczekiwanie wątku na wywołanie metody `notify()` lub `notifyAll()`, istnieje bardzo rzadko spotykana możliwość, że wątek zakończy oczekiwanie na skutek tak zwanego **fałszywego przebudzenia**. Omówienie skomplikowanych warunków, które prowadzą do takiej sytuacji, wykracza poza zakres tej książki. Ze względu na możliwość wystąpienia fałszywego przebudzenia wątku firma Oracle zaleca wywoływanie metody `wait()` w pętli sprawdzającej warunek, na którego spełnienie oczekuje wątek. W następnym przykładzie zastosujemy tę technikę.

Przykład użycia metod `wait()` i `notify()`

Aby łatwiej Ci było zrozumieć zastosowania metod `wait()` i `notify()`, stworzymy teraz program symulujący tykanie zegara przez wyświetlanie na przemian słów „tik” i „tak”. W tym celu stworzymy klasę `TickTock` zawierającą dwie metody: `tick()` i `tock()`. Metoda `tick()` wyświetli słowo „tik”, a metoda `tock()` słowo „tak”. Aby uruchomić symulację zegara, utworzymy dwa wątki, z których jeden będzie wywoływać metodę `tick()`, a drugi metodę `tock()`. Wątki te muszą być wykonywane w taki sposób, aby program wyświetlał na przemian słowa „tik” i „tak”. Kod źródłowy programu przedstawiłem na listingu 11.10.

Listing 11.10. `ThreadCom.java`

```
// Używa metod wait() i notify()
// do stworzenia symulacji zegara.

class TickTock {
```

```

String state; //przechowuje stan zegara

synchronized void tick(boolean running) {
    if(!running) { //zatrzymuje zegar
        state = "ticked";
        notify(); //powiadamia oczekujący wątek
        return;
    }

    System.out.print("tik ");

    state = "ticked"; //zmienia stan zegara na "ticked"

    notify(); //umożliwia wykonanie metody tock() ← Metoda tick() powiadamia metodę tock().
    try {
        while(!state.equals("tocked"))
            wait(); //oczekuje na zakończenie metody tock() ← Metoda tick() czeka na zakończenie
    }
    catch(InterruptedException exc) {
        System.out.println("Wątek został przerwany.");
    }
}

synchronized void tock(boolean running) {
    if(!running) { //zatrzymuje zegar
        state = "tocked";
        notify(); //powiadamia oczekujący wątek
        return;
    }

    System.out.println("tak");

    state = "tocked"; //zmienia stan zegara na "tocked"

    notify(); //umożliwia wykonanie metody tick() ← Metoda tock() powiadamia metodę tick().
    try {
        while(!state.equals("ticked"))
            wait(); //oczekuje na zakończenie metody tick() ← Metoda tock() czeka
    }
    catch(InterruptedException exc) {
        System.out.println("Wątek został przerwany.");
    }
}
}

class MyThread implements Runnable {
    Thread thrd;
    TickTock ttOb;

    //Tworzy nowy wątek.
    MyThread(String name, TickTock tt) {
        thrd = new Thread(this, name);
        ttOb = tt;
        thrd.start(); //uruchamia wątek
    }
    //Rozpoczyna wykonywanie nowego wątku.

```

```

public void run() {
    if(thrd.getName().compareTo("tik") == 0) {
        for(int i=0; i<5; i++) ttOb.tick(true);
        ttOb.tick(false);
    }
    else {
        for(int i=0; i<5; i++) ttOb.tock(true);
        ttOb.tock(false);
    }
}
}

class ThreadCom {
    public static void main(String args[]) {
        TickTock tt = new TickTock();
        MyThread mt1 = new MyThread("tik", tt);
        MyThread mt2 = new MyThread("tak", tt);

        try {
            mt1.thrd.join();
            mt2.thrd.join();
        } catch(InterruptedException exc) {
            System.out.println("Wątek główny został przerwany.");
        }
    }
}

```

A oto wynik działania tego programu:

```

tik tak
tik tak
tik tak
tik tak
tik tak

```

Przyjrzyjmy się bliżej działaniu tego programu. Sercem symulacji zegara jest klasa `TickTock`. Zawiera ona dwie metody, `tick()` i `tock()`, które komunikują się ze sobą, aby zagwarantować, że po metodzie `tick()` zostanie zawsze wykonana metoda `tock()`, a po niej znowu metoda `tick()` i tak dalej. Zwróć uwagę na składową `state`. Gdy działa symulacja zegara, składowa `state` przechowuje łańcuch "ticked" lub "tocked" informujący o bieżącym stanie zegara. Metoda `main()` tworzy obiekt `tt` klasy `TickTock` i używa go do uruchomienia dwóch wątków potomnych.

Wątki te opierają się na obiektach typu `MyThread`. Konstruktor klasy `MyThread` ma dwa parametry. Pierwszym jest nazwa wątku, "tik" lub "tak". Drugim jest referencja obiektu typu `TickTock`, `tt` w tym przypadku. Wewnątrz metody `run()` klasy `MyThread` wywołana zostaje metoda `tick()`, jeśli nazwą wątku jest "tik". Jeśli nazwą jest "tak", wywołana zostaje metoda `tock()`. Każda z tych metod zostaje wywołana pięć razy z argumentem `true`. Ostatnie wywołanie przekazuje każdej metodzie argument `false`, co powoduje zatrzymanie symulacji zegara.

Najważniejszą część symulacji zegara stanowią metody `tick()` i `tock()` klasy `TickTock`. Przyjrzyjmy się metodzie `tick()`, której kod przedstawiłem poniżej.

```

synchronized void tick(boolean running) {
    if(!running) { //zatrzymuje zegar
        state = "ticked";
        notify(); //powiadamia oczekujący wątek
        return;
    }

    System.out.print("tik ");

    state = "ticked"; //zmienia stan zegara na "ticked"

    notify(); //umożliwia wykonanie metody tock()
    try {
        while(!state.equals("tocked"))
            wait(); //oczekuje na zakończenie metody tock()
    }
    catch(InterruptedException exc) {
        System.out.println("Wątek został przerwany.");
    }
}

```

Zwróć uwagę, że metoda `tick()` została zadeklarowana jako `synchronized`. Jak już wspominałem, metody `wait()` i `notify()` znajdują zastosowanie tylko w przypadku synchronizacji dostępu do metod. Metoda rozpoczyna działanie od sprawdzenia wartości parametru `running`. Parametr ten jest używany do zakończenia działania symulacji zegara. Jeśli ma wartość `false`, zegar zostaje zatrzymany. W takim przypadku składowa `state` otrzymuje wartość `"ticked"` i wywołana zostaje metoda `notify()`, aby umożliwić działanie oczekującego wątku. Za chwilę wrócimy jeszcze do tego punktu.

Jeśli zegar działa, metoda wyświetla słowo `"tik"`, nadaje składowej `state` wartość `"ticked"` i wywołuje metodę `notify()`. Wywołanie metody `notify()` umożliwia wznowienie działania wątku, który oczekiwał na dostęp do tego samego obiektu. Następnie w pętli `while` wywoływana jest metoda `wait()`. Wywołanie metody `wait()` powoduje zawieszenie wykonywania metody `tick()` do momentu, w którym inny wątek wywoła metodę `notify()`. Zatem wykonanie pętli zostanie również wstrzymane do momentu, w którym inny wątek wywoła metodę `notify()` dla tego samego obiektu. W rezultacie wywołanie metody `tick()` powoduje wyświetlenie jednego słowa `"tik"`, wznowienie innego wątku i zawieszenie własnego wykonania.

Pętla `while`, która wywołuje metodę `wait()`, sprawdza, czy składowa `state` ma wartość `"tocked"`, co zdarzy się dopiero po wywołaniu metody `tock()`. Jak już wcześniej wyjaśniłem, zastosowanie pętli `while` do sprawdzania tego warunku ma zapobiec fałszywemu przebudzeniu wątku. Gdy metoda `wait()` zwróci sterowanie, a składowa `state` nie ma wartości `"tocked"`, oznacza to właśnie wystąpienie fałszywego przebudzenia i metoda `wait()` zostaje wywołana ponownie.

Metoda `tock()` stanowi dokładną kopię metody `tick()`, z tą różnicą, że wyświetla słowo `"tak"` i nadaje składowej `state` wartość `"tocked"`. Zatem metoda `tock()` najpierw wyświetla słowo `"tak"`, wywołuje metodę `notify()` i następnie zawiesza swoje działanie. Metody `tick()` i `tock()` mogą działać tylko razem, przy czym po wywołaniu metody `tick()` musi nastąpić wywołanie metody `tock()`, po którym znowu następuje wywołanie metody `tick()` i tak dalej. Działanie tych metod jest wzajemnie zsynchronizowane.

Po zatrzymaniu zegara metoda `notify()` zostaje wywołana ostatni raz, aby wznowić działanie drugiego wątku. Pamiętaj, że metody `tick()` i `tock()` wywołują zawsze metodę `wait()` po wyświetleniu komunikatu. Zatem gdy zegar zostanie zatrzymany, jeden z wątków będzie się znajdował w stanie oczekiwania. Jego działanie można wznowić właśnie za pomocą ostatniego wywołania metody `notify()`. Spróbuj usunąć to wywołanie metody `notify()` i zaobserwować efekt. Zobaczysz, że program „zawiesi się” i będziesz musiał przerwać jego działanie, naciskając `Ctrl+C`. Dzieje się tak, ponieważ gdy ostatnie wywołanie metody `tock()` spowoduje wywołanie metody `wait()`, to nie będzie odpowiadającego mu wywołania metody `notify()`, które pozwoliłoby metodzie `tock()` zakończyć działanie. Zatem metoda `tock()` oczekuje wtedy w nieskończoność.

Jeśli masz jeszcze wątpliwości, czy rzeczywiście wywołania metod `wait()` i `notify()` są konieczne dla prawidłowego działania symulacji zegara, zastąp klasę `TickTock` jej poniższą wersją, z której usunąłem wywołania metod `wait()` i `notify()`.

```
// Bez wywołań metod wait() i notify().

class TickTock {

    String state; // przechowuje stan zegara

    synchronized void tick(boolean running) {
        if(!running) { // zatrzymuje zegar
            state = "ticked";
            return;
        }

        System.out.print("tik ");

        state = "ticked"; // zmienia stan zegara na "ticked"
    }

    synchronized void tock(boolean running) {
        if(!running) { // zatrzymuje zegar
            state = "tocked";
            return;
        }

        System.out.println("tak");

        state = "tocked"; // zmienia stan zegara na "tocked"
    }
}
```

Po zastąpieniu klasy `TickTock` nową wersją program wyświetli poniższe komunikaty:

```
tik tik tik tik tik tak
tak
tak
tak
tak
```

Metody `tick()` i `tock()` nie współpracują już ze sobą!

Ekspert odpowiada

Pytanie: Spotkałem się z terminem *zakleszczenie* w odniesieniu do nieprawidłowo działających programów wielowątkowych. Na czym polega ta sytuacja i jak jej unikać? Na czym polega *sytuacja wyścigu* i jak mogę jej uniknąć?

Odpowiedź: Zakleszczenie, jak wskazuje nazwa, jest sytuacją, w której jeden wątek czeka, aż drugi wątek wykona pewną operację, ale drugi wątek czeka właśnie na pierwszy. Działanie obu wątków jest wstrzymane, wątki oczekują na siebie wzajemnie. Przypomina to trochę dwie nadzwyczaj uprzejme osoby, które nie mogą przekroczyć progu, ponieważ każda chce ustąpić drugiej.

Unikanie zakleszczeń wydaje się proste, ale takie nie jest. Sama analiza kodu programu często nie wystarcza do ustalenia przyczyny zakleszczenia, ponieważ w czasie wykonania programu między wątkami mogą zachodzić skomplikowane zależności czasowe. Unikanie zakleszczeń wymaga dbałości w programowaniu oraz wszechstronnego testowania. Pamiętaj, że jeśli program wielowątkowy czasami zawiesza się, to najbardziej prawdopodobną tego przyczyną jest właśnie zakleszczenie.

Sytuacja wyścigu występuje, gdy dwa (lub więcej) wątków próbuje uzyskać równocześnie dostęp do współdzielonego zasobu, ale odbywa się to bez odpowiedniej synchronizacji. Na przykład jeden wątek może przypisywać zmiennej nową wartość, podczas gdy inny wątek w tym samym czasie zwiększa wartość tej samej zmiennej. Jeśli nie zastosujemy synchronizacji, to wynik tych operacji będzie zależał od kolejności wykonania wątków (czy drugi wątek najpierw zwiększy oryginalną wartość zmiennej, czy raczej pierwszy wątek zdąży przypisać jej wcześniej nową wartość?). W takich sytuacjach mówi się o wyścigu wątków, ponieważ ostateczny wynik operacji zależy od tego, który wątek będzie pierwszy. Podobnie jak w przypadku zakleszczenia wykrycie sytuacji wyścigu może być trudne. Najlepszym rozwiązaniem jest zapobieganie: staranne programowanie z zastosowaniem odpowiedniej synchronizacji dostępu do współdzielonych zasobów.

Wstrzymywanie, wznowianie i kończenie działania wątków

Możliwość wstrzymania wykonywania wątku bywa przydatna w pewnych sytuacjach. Na przykład osobny wątek może wyświetlać czas. Jeśli użytkownik nie chce oglądać zegara, wtedy można wstrzymać działanie wątku, który go wyświetla. Niezależnie od przyczyny wstrzymanie wykonywania wątku nie powinno stanowić problemu. Podobnie jak późniejsze wznowienie jego działania.

Mechanizm wstrzymywania, wznowiania i kończenia działania wątków różni się we wcześniejszych i obecnych wersjach Javy. W wersjach wcześniejszych niż Java 2 do wstrzymywania, wznowiania i kończenia działania wątków służyły odpowiednio metody `suspend()`, `resume()` i `stop()` zdefiniowane w klasie `Thread`. Mają one następującą postać:

```
final void resume( )
final void suspend( )
final void stop( )
```

Mimo że metody te wydają się właściwym rozwiązaniem kwestii zarządzania wykonaniem wątków, obecnie ich użycie nie jest zalecane. W Javie 2 uznano metodę `suspend()` klasy `Thread` za przestarzałą. Powodem była możliwość doprowadzenia w pewnych sytuacjach do

zakleszczeń na skutek użycia tej metody. Metodę `resume()` również uznano za przestarzałą. Nie powodowała ona żadnych problemów, ale jej użycie bez metody `suspend()` nie miało sensu. Metodę `stop()` również uznano za przestarzałą w Javie 2, ponieważ w pewnych warunkach jej użycie mogło być przyczyną poważnych problemów.

Skoro nie można już używać metod `suspend()`, `resume()` i `stop()` do sterowania działaniem wątku, mogłoby się wydawać, że nie ma sposobu, by wstrzymać, wznowić lub zakończyć wykonywanie wątku. Nic bardziej błędnego. Wystarczy zaprojektować wątek w taki sposób, aby metoda `run()` sprawdzała okresowo, czy powinna wstrzymać, wznowić lub zakończyć swoje działanie. Zwykle osiąga się to przez wprowadzenie w klasie wątków dwóch znaczników: jednego dla wstrzymania i wznowienia wątku i drugiego dla zatrzymania wątku. W przypadku operacji wstrzymania i wznowienia wątku, jeśli odpowiedni znacznik nie jest ustawiony, metoda `run()` musi kontynuować wykonywanie wątku. W przeciwnym razie musi wstrzymać swoje działanie. Jeśli ustawiony jest drugi ze znaczników, oznaczający zatrzymanie wątku, metoda `run()` musi zakończyć działanie.

Na listingu 11.11 przedstawiłem jeden ze sposobów implementacji własnych wersji metod `suspend()`, `resume()` i `stop()`.

Listing 11.11. *Suspend.java*

```
// Wstrzymywanie, wznowianie i zatrzymywanie wątku.

class MyThread implements Runnable {
    Thread thrd;
    boolean suspended; ← Powoduje wstrzymanie wykonania wątku, gdy ma wartość true.
    boolean stopped; ← Powoduje zakończenie wykonania wątku, gdy ma wartość true.

    MyThread(String name) {
        thrd = new Thread(this, name);
        suspended = false;
        stopped = false;
        thrd.start();
    }

    // Punkt wejścia do wątku.
    public void run() {
        System.out.println(thrd.getName() + " rozpoczyna działanie.");
        try {
            for(int i = 1; i < 1000; i++) {
                System.out.print(i + " ");
                if((i%10)==0) {
                    System.out.println();
                    Thread.sleep(250);
                }

                // Używa bloku synchronized, aby sprawdzić
                // wartości składowych suspended i stopped.
                synchronized(this) { ← Ten blok synchronized sprawdza wartości składowych
                    while(suspended) { suspended i stopped.
                        wait();
                    }
                    if(stopped) break;
                }
            }
        }
    }
}
```

```

    }
    } catch (InterruptedException exc) {
        System.out.println(thrd.getName() + " został przerwany.");
    }
    System.out.println(thrd.getName() + " kończy działanie.");
}

// Zatrzymuje wątek.
synchronized void mystop() {
    stopped = true;

    // Poniższe instrukcje umożliwiają zatrzymanie wątku,
    // którego wykonanie zostało wstrzymane.
    suspended = false;
    notify();
}

// Wstrzymuje działanie wątku.
synchronized void mysuspend() {
    suspended = true;
}

// Wznawia działanie wątku.
synchronized void myresume() {
    suspended = false;
    notify();
}
}

class Suspend {
    public static void main(String args[]) {
        MyThread ob1 = new MyThread("Wątek potomny klasy MyThread");

        try {
            Thread.sleep(1000); // umożliwia rozpoczęcie wykonywania wątku ob1

            ob1.mysuspend();
            System.out.println("Wstrzymuję wątek.");
            Thread.sleep(1000);

            ob1.myresume();
            System.out.println("Wznawiam wątek.");
            Thread.sleep(1000);

            ob1.mysuspend();
            System.out.println("Wstrzymuję wątek.");
            Thread.sleep(1000);

            ob1.myresume();
            System.out.println("Wznawiam wątek.");
            Thread.sleep(1000);

            ob1.mysuspend();
            System.out.println("Zatrzymuję wątek.");
            ob1.mystop();
        }
    }
}

```

```
    } catch (InterruptedException e) {
        System.out.println("Wątek główny został przerwany");
    }

    // czeka na zakończenie wątku
    try {
        obl.thrd.join();
    } catch (InterruptedException e) {
        System.out.println("Wątek główny został przerwany");
    }

    System.out.println("Wątek główny kończy działanie.");
}
}
```

Poniżej przedstawiłem rezultat działania programu (może nieco różnić się w Twoim przypadku).

```
Wątek potomny klasy MyThread rozpoczyna działanie.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Wstrzymuję wątek.
Wznawiam wątek.
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Wstrzymuję wątek.
Wznawiam wątek.
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120
Zatrzymuję wątek.
Wątek potomny klasy MyThread kończy działanie.
Wątek główny kończy działanie.
```

Program ten działa w następujący sposób. Klasa `MyThread` definiuje dwie zmienne typu `boolean` o nazwach `suspended` i `stopped`, które zarządzają operacjami wstrzymania i zatrzymania wątku. Konstruktor klasy `MyThread` nadaje tym zmiennym wartość początkową `false`. Metoda `run()` zawiera blok `synchronized`, który sprawdza wartość zmiennej `suspended`. Jeśli jest ona równa `true`, wywołuje metodę `wait()`, aby wstrzymać działanie wątku. Aby wstrzymać działanie wątku, wywołujemy metodę `mysuspend()`, która nadaje zmiennej `suspended` wartość `true`. Aby wznowić wątek, wywołujemy metodę `myresume()`, która nadaje zmiennej `suspended` wartość `false` i wywołuje metodę `notify()`.

Aby zakończyć wątek, wywołujemy metodę `mystop()`, która nadaje zmiennej `stopped` wartość `true`. Dodatkowo metoda `mystop()` nadaje zmiennej `suspended` wartość `false`, a następnie wywołuje metodę `notify()`. Jest to konieczne, aby zakończyć wątek, którego wykonanie zostało wcześniej wstrzymane.

Ekspert odpowiada

Pytanie: Wielowątkowość wydaje się być doskonałym sposobem poprawy efektywności moich programów. Czy możesz podać wskazówkę, jak najefektywniej korzystać z tego mechanizmu?

Odpowiedź: Klucz do efektywnego wykorzystania wielowątkowości polega na przestawieniu się z myślenia kategoriami sekwencyjnego działania na bardziej równoległe. Na przykład gdy w Twoim programie istnieją dwa podsystemy działające zupełnie niezależnie od siebie, powinieneś poważnie rozważyć możliwość zaimplementowania ich jako osobnych wątków. Jedno słowo przestrogi. Jeśli stworzysz zbyt wiele wątków, możesz pogorszyć efektywność działania programu, zamiast ją poprawić. Pamiętaj o narzucie związanym z przełączaniem kontekstu wykonania wątków. Jeśli Twój program uruchomi zbyt wiele wątków, większość czasu procesora zajmie przełączanie kontekstu wątków, a nie wykonanie programu!

Przykład 11.2. Wykorzystanie głównego wątku*UseMain.java*

Wszystkie programy w Javie mają przynajmniej jeden wątek wykonania zwany **wątkiem głównym**. Wątek ten powstaje automatycznie w momencie uruchomienia programu. Jak dotąd zaniedbywaliśmy istnienie wątku głównego, ale w tym przykładzie przekonasz się, że można się nim posługiwać tak jak każdym innym wątkiem.

1. Utwórz plik o nazwie *UseMain.java*.
2. Dostęp do wątku głównego wymaga uzyskania obiektu typu `Thread` reprezentującego ten wątek. W tym celu wywołamy metodę `currentThread()` zadeklarowaną jako `static` w klasie `Thread`. Jej ogólną postać przedstawiłem poniżej:

```
static Thread currentThread()
```

Metoda ta zwraca referencję wątku, w którym została wywołana. Zatem jeśli wywołamy ją w głównym wątku, otrzymamy referencję głównego wątku. Dysponując tą referencją, możemy kontrolować działanie głównego wątku w taki sam sposób jak innych wątków.

3. Wprowadź w pliku tekst źródłowy programu przedstawiony na listingu 11.12. Program ten pobiera referencję wątku głównego, a następnie zmienia jego nazwę i priorytet.

Listing 11.12. UseMain.java

```
/*
   Przykład 11.2

   Sterowanie wątkiem głównym.
*/

class UseMain {
    public static void main(String args[]) {
        Thread thrd;

        // Pobiera referencję wątku głównego.
        thrd = Thread.currentThread();

        // Wyświetla nazwę wątku głównego.
        System.out.println("Nazwa wątku głównego: " +
            thrd.getName());
    }
}
```

```
// Wyświetla priorytet wątku głównego.
System.out.println("Priorytet wątku głównego: " +
    thrd.getPriority());

System.out.println();

// Konfiguruje nazwę i priorytet wątku głównego.
System.out.println("Zmieniam nazwę i priorytet wątku głównego.\n");
thrd.setName("Wątek nr 1");
thrd.setPriority(Thread.NORM_PRIORITY+3);

System.out.println("Aktualna nazwa wątku głównego: " +
    thrd.getName());

System.out.println("Bieżący priorytet wątku głównego: " +
    thrd.getPriority());
}
}
```

4. Poniżej przedstawiłem wynik działania tego programu:

```
Nazwa wątku głównego: main
Priorytet wątku głównego: 5

Zmieniam nazwę i priorytet wątku głównego.

Aktualna nazwa wątku głównego: Wątek nr 1
Bieżący priorytet wątku głównego: 8
```

5. Wykonując operacje na wątku głównym, powinieneś zachować ostrożność. Na przykład: jeśli umieścisz na końcu metody `main()` kod przedstawiony poniżej, to program nigdy nie zakończy swojego działania, ponieważ będzie oczekiwać na zakończenie wątku głównego!

```
try {
    thrd.join();
} catch (InterruptedException exc) {
    System.out.println("Wątek przerwany");
}
```

Test sprawdzający

1. W jaki sposób wielowątkowość w Javie umożliwia tworzenie efektywniejszych programów?
2. Wielowątkowość w Javie opiera się na klasie _____ i interfejsie _____.
3. Dlaczego czasami, tworząc wątki, lepiej jest utworzyć klasę pochodną klasy `Thread`, niż implementować interfejs `Runnable`?
4. Pokaż, jak użyć metody `join()` do oczekiwania na zakończenie wątku `MyThrd`.

5. Zwiększ priorytet wątku `MyThrd` o trzy poziomy ponad normalny.
6. Na czym polega efekt dodania słowa kluczowego `synchronized` w deklaracji metody?
7. Metod `wait()` i `notify()` używamy do _____.
8. Zmodyfikuj klasę `TickTock` w taki sposób, aby rzeczywiście symulowała upływ czasu. Zatem niech każdy „tik” trwa pół sekundy, podobnie „tak”. Każdy tik-tak zajmie zatem jedną sekundę (nie przejmuj się czasem potrzebnym na przełączenie wątku).
9. Dlaczego w nowych programach nie należy używać metod `suspend()`, `resume()` i `stop()`?
10. Jaka metoda klasy `Thread` zwraca nazwę wątku?
11. Co zwraca metoda `isAlive()`?
12. Spróbuj samodzielnie wprowadzić synchronizację w klasie `Queue` stworzonej w poprzednich rozdziałach w taki sposób, aby zapewnić jej bezpieczne użycie w wielu wątkach.

Skorowidz

A

- abstrakcyjna klasa bazowa, 284
- adnotacje, 400, 422
 - ogólnego przeznaczenia, 424
 - znacznikowe, 424
- algorytm sortowania Quicksort, 212, 216
- anonimowa klasa wewnętrzna, 221, 521
- API, Application Programming Interface, 279
- aplet, 22, 464, 467
 - bazujący na AWT, 470
 - MySwingApplet, 524
 - obsługujący zdarzenia myszy, 483
 - Swing, 522
 - wyświetlający baner, 471
- aplety
 - architektura, 467
 - działanie, 469
 - parametry, 475
 - szkielet, 468
- architektura
 - model-delegat, 493
 - model-widok-nadzorca, 493
 - z wydzielonym modelem, 493
- argument, 127
- argument wieloznaczny, 438
- ASCII, 53, 339
- asercje, 488
- automatyczne
 - konwersje typów, 71
 - opakowywanie, 413, 415, 418
 - wypakowywanie, 413
 - zarządzanie zasobami, 312, 336
- AWT, Abstract Window Toolkit, 464, 492

B

- bajt, 334
- bezpieczeństwo, 23, 25
- biblioteka klas, 48, 279

- biblioteka Swing, 464, 491
- bit, 176
- bitowe operatory przypisania, 181
- blok
 - finally, 309, 332
 - kodu, 43
 - static, 215
 - synchronized, 384
 - try, 299, 304, 312, 332
- blokada, 381
- błąd
 - otwarcia pliku, 334
 - niejednoznaczności, 227, 457
 - przekroczenia zakresu, 300
 - składni, 34
 - w kodzie programu, 297
 - wewnętrzne, 297
- bufor, 80, 353
- buforowanie wierszy, 80

C

- ciało metody, 123
- czas istnienia zmiennej, 62

D

- definiowanie klasy, 32, 117
- definiowanie pakietu, 270
- deklaracja
 - bloku static, 215
 - klasy generycznej, 434
 - klasy pochodnej, 232
 - tablicy, 146, 155
 - zmiennej, 36
- dekoder, 353
- dekrementacja, 42, 65
- delegacja zdarzeń, 481
- delegat interfejsu użytkownika, 493
- destruktor, 140

dodawanie konstruktora do klasy, 137
 dopełnienie do dwóch, 180
 dostęp
 do pakietu, 274
 do pliku, 343
 do składowych, 190, 232, 273
 do składowych klasy bazowej, 240
 do systemu, 34
 sekwencyjny, 343
 drzewo hierarchii, 579
 dynamiczne zarządzanie pamięcią, 139
 dynamiczność, 25
 dynamiczny wybór metody, 255
 dziedziczenie, 28, 230, 232, 235, 241
 dziedziczenie wielobazowe, 232
 dzielenie przez zero, 68, 185

E

Eclipse, 29
 etykieta, 104
 etykieta done, 105
 etykieta jlabContents, 509

F

falszywe przebudzenie, 387
 funkcje wirtualne, 256

G

generowanie wyjątku, 305
 generyczna klasa Queue, 448
 generyczność, 427
 graficzny interfejs użytkownika, 467

H

hermetyzacja, 27
 hermetyzowanie tablicy, 192
 hierarchia
 dziedziczenia, 28
 klas, 230, 241
 wielopoziomowa klas, 244
 wyjątków, 296
 zawierania, 494

I

IDE, 29
 identyfikator, 47
 implementacja, 279
 implementacja interfejsu, 281
 import składowej Static.out, 422
 import składowych statycznych, 400, 420

importowanie pakietu, 278
 inicjalizacja, 90
 dynamiczna, 60
 obiektu, 206
 tablic wielowymiarowych, 153
 zmiennej, 59
 inicjalizator, 62
 inkrementacja, 42, 65
 instrukcja
 break, 86, 102
 break z etykieta, 104
 continue, 108
 continue z etykieta, 108
 default, 85, 89
 for, 41
 goto, 104
 if, 40, 81
 if-else-if, 83
 import, 277
 import static, 422
 new, 120, 136
 package, 270
 println(), 36
 return, 124
 super(), 238
 switch, 84, 85, 172
 throw, 297
 try, 338
 instrukcje
 iteracji, 79
 skoku, 79
 wyboru, 79
 interfejs, 269, 279
 ActionListener, 504
 Annotation, 423
 AutoCloseable, 336
 Closeable, 336
 Containment, 447
 DataInput, 339
 ListSelectionListener, 514
 ListSelectionModel, 513
 MouseListener, 482
 MouseMotionListener, 482
 Queue, 286
 Runnable, 366
 Series, 283
 interfejsy
 dziedziczące, 291
 generyczne, 445
 słuchaczy zdarzeń, 482
 interpreter kodu bajtowego, 24, 31
 interpretowalność, 25
 iteracja, 41

J

J2SE, Java 2 Standard Edition, 14
Java API, 279
Java SE, 15
JDK, Java Development Kit, 17, 29
język C i C++, 21
język C#, 22
JFC, Java Foundation Classes, 492
JIT, just-in-time, 24
JVM, Java Virtual Machine, 24

K

kanal, 353
klasa, 27, 115
 ActionEvent, 503, 504
 Applet, 477
 ArithmeticException, 300
 Book, 274
 BookDemo, 274
 BufferedReader, 348, 352
 ByThrees, 283
 ByTwos, 281
 CircularQueue, 287
 Component, 470, 495
 Console, 345
 Container, 495
 DataInputStream, 339
 DataOutputStream, 339
 Error, 296
 ErrorInfor, 199
 ErrorMsg, 265
 Exception, 296
 FailSoftArray, 192
 FileInputStream, 330
 FileOutputStream, 330, 334
 FileReader, 351
 FileWriter, 350
 GenericMethodDemo, 443
 Help, 130
 InputStream, 328
 InputStreamReader, 352
 ItemEvent, 509
 JApplet, 522
 java.lang.Enum, 406
 JButton, 506
 JCheckBox, 509
 JComponent, 494
 JList, 512
 JTextField, 506
 ListSelectionEvent, 513
 Math, 53, 420
 MyThread, 367, 395

NonIntResultException, 316
NumericFns, 435
Object, 267
OutputStreamWriter, 351
Pair, 437
PrintWriter, 349
Pwr, 143
Queue, 158, 194, 207, 448
Quicksort, 217
RandomAccessFile, 343
Reader, 352
RuntimeException, 313
Scanner, 360
ShowBits, 182, 221
SimpleApplet, 465
String, 167
StringBuffer, 171
Summation, 445
Sup, 256
System, 48, 279, 326
Thread, 365, 374
Throwable, 296, 307, 315
TickTock, 389
Timer, 212
Triangle, 233, 250
TwoDShape, 231, 250, 264
Vehicle, 119
Writer, 351
klasy
 abstrakcyjne, 260
 bazowe, 229
 generyczne, 428
 komponentów Swing, 495
 nadrzędne, 28
 NIO, 353
 opakowujące, 353–355
 pochodne, 229, 232
 pochodne klasy Exception, 315
 pochodne klasy Thread, 370
 strumieni bajtowych, 325
 strumieni znakowych, 326
 wewnętrzne, 219, 521
 zagnieżdżone, 218
 zagnieżdżone static, 222
 zdarzeń, 481
klauzula
 case, 85
 catch, 298, 302, 312
 else, 81
 extends, 435, 440
 finally, 309
 implements, 281
 throws, 311
 throws java.io.IOException, 80

kod bajtowy, 24
 koder, 353
 kodowanie, 31
 kolejka, 158
 kolejka cykliczna, 288
 kolejka dynamiczna, 286
 komentarz, 32

- dokumentacyjny, 573, 578
- jednowierszowy, 33
- wielowierszowy, 32

 kompilator, 28
 kompilator javac, 31, 32
 kompilator JIT, 24
 kompilowanie programu, 24, 31
 komponent

- JButton, 502
- JCheckBox, 509
- JList, 512
- JTextField, 506

 komponenty Swing, 494
 komunikacja międzywątkowa, 386
 konsola znakowa, 80
 konstruktor, 135

- domyślny, 238
- generyczny, 445
- klasy bazowej, 235, 238
- klasy Queue, 207
- klasy Thread, 369
- klasy Throwable, 315
- klasy Triangle, 236
- klasy TwoDShape, 237
- MyClass(), 206
- o jednym parametrze, 238
- Transport(), 405
- TwoDShape(), 252
- typu wyliczeniowego, 404
- z parametrem, 136

 kontekst graficzny, 465
 kontener JApplet, 522
 kontener JFrame, 522
 kontenery szczytowe, 495
 konwersja typów w wyrażeniach, 76
 konwersja typu, 71
 kopiowanie pliku, 335
 kopiowanie tablicy, 158

L

LIFO, last-in, first-out, 28
 lista argumentów o zmiennej długości, 221
 lista JList, 512
 literały, 56

- łańcuchowe, 58
- znakowe, 59

Ł

łańcuch wyjątków, 315
 łańcuch znaków, 58, 167

M

maszyna wirtualna Java, 24
 menedżer

- BorderLayout, 501
- FlowLayout, 504, 505
- układu, 496

 metadane, 422
 metoda, 27, 33, 122
 metoda

- abs(), 205
- absEqual(), 438
- actionPerformed(), 509
- addActionListener(), 506, 521
- area(), 261
- charAt(), 171
- close(), 331, 334
- compareTo(), 355, 406
- console(), 345
- destroy(), 470
- doubleValue(), 414, 435
- finalize(), 139
- floatValue(), 414
- generator(), 141
- generyczna arraysEqual(), 443
- genException(), 307
- get(), 286
- getActionCommand(), 503
- getClass(), 267
- getContentPane(), 500
- getErrorInfo(), 199
- getErrorMsg(), 199
- getob(), 430
- getParameter(), 475
- getSelectedIndex(), 513
- getSelectedIndices(), 514
- getSpeed(), 405
- getText(), 506
- init(), 469, 485
- invokeAndWait(), 501, 502
- invokeLater(), 501, 502
- isAlive(), 375
- isFactor(), 128
- join(), 375
- main(), 33
- makeGUI(), 524
- Math.abs(), 438
- Math.pow(), 420
- mouseClicked(), 482
- mouseDragged(), 482

- mouseEntered(), 482
 - mouseExited(), 482
 - mouseMoved(), 482
 - noChange(), 197
 - notify(), 386
 - notifyAll(), 386
 - ordinal(), 406
 - ovlDemo(), 202
 - paint(), 465, 470
 - parseDouble(), 354
 - print(), 37, 349
 - println(), 34, 37, 349, 352
 - printStackTrace(), 308
 - prompt(), 311
 - put(), 286
 - range(), 122
 - read(), 80, 328
 - readLine(), 348
 - readPassword(), 345
 - reciprocal(), 435
 - removeActionListener(), 506
 - removeKeyListener(), 480
 - repaint(), 470
 - resume(), 392
 - run(), 366, 393
 - setActionCommand(), 506, 508
 - setCharAt(), 171
 - setName(), 369
 - setPriority(), 378
 - show(), 253
 - showStatus(), 475
 - showType(), 431
 - sleep(), 367
 - sqrt(), 53, 214
 - start(), 469
 - static, 214
 - stop(), 392
 - substring(), 171
 - sumArray(), 384
 - suspend(), 392
 - System.console(), 345
 - tick(), 387
 - tock(), 387
 - toString(), 308, 414
 - update(), 471
 - valueChanged(), 514
 - values(), 403
 - valuesOf(), 403
 - vaTest(), 223
 - vaTest(int ...), 227
 - wait(), 386, 391
 - write(), 334
 - metody
 - abstrakcyjne, 261, 279
 - dostępowe dla składowych prywatnych, 233
 - generyczne, 443
 - klasy Applet, 477
 - klasy InputStream, 327
 - klasy Object, 267
 - klasy OutputStream, 328
 - klasy Reader, 346
 - klasy Thread, 365
 - klasy Throwable, 308
 - klasy Writer, 347
 - konwersji łańcuchów, 354
 - natywne, 488
 - o zmiennej liczbie argumentów, 222
 - rekurencyjne, 210
 - pęcherzykowego, 216
 - wejścia klasy DataInputStream, 340
 - wyjścia klasy DataOutputStream, 339
 - model, 493
 - model delegacji zdarzeń, 479
 - modyfikator
 - abstract, 261
 - dostępu, 33
 - final, 312
 - private, 33, 191, 194, 232
 - protected, 190, 275
 - public, 33, 190
 - transient, 486
 - volatile, 486
 - modyfikatory dostępu, 190
 - monitor, 381
- ## N
- nadzorca, 493
 - najmłodszy bit, 180
 - narzut, 420
 - nawias klamrowy, 34
 - nawiasy, 77
 - nazwa
 - klasy, 138
 - obiektu, 118
 - pliku źródłowego, 30
 - NetBeans, 29
 - niejednoznaczność, 226
 - niezależność, 25
 - niezawodność, 25
- ## O
- obiekt, 27, 116
 - ActionEvent, 503
 - klasy String, 167
 - przekazywany metodzie, 195
 - System.err, 327
 - System.in, 327

obiekt
 System.out, 327
 Thread, 369
 typu Integer, 431
 typu T, 437
 zablokowany, 381
 obiektowość, 25
 obliczanie silni, 210
 obsługa
 błędów, 301, 317
 błędów wejścia-wyjścia, 331
 plików, 350
 wejścia konsoli, 328
 wyjątków, 295
 wyjścia konsoli, 329, 349
 zdarzeń, 479
 zdarzeń myszy, 482
 odczyt konsoli, 345
 odczyt z pliku, 330
 odczyt znaków, 346
 odstępy, 77
 odwołanie do obiektu, 120
 odzyskiwanie nieużytków, 139
 ogólna klasa akcji, 28
 ograniczanie argumentów wieloznacznych, 440
 ograniczanie typów, 434
 ograniczenia dla składowych statycznych, 458
 ograniczenia tablic generycznych, 459
 ograniczenia związane z wyjątkami, 460
 okno
 apletu SimpletApplet, 466
 programu ListDemo, 515
 programu SwingFC, 516
 programu TFDemo, 508
 statusu, 475
 opakowywanie, 414
 opcja classpath, 271
 operacja bitowa z operatorem przypisania, 181
 operacja get, 159
 operacja put, 159
 operacje na łańcuchach, 168
 operator, 63
 ?, 184
 +, 36
 bitowy AND, 175
 bitowy NOT, 179
 bitowy OR, 177
 bitowy XOR, 178
 delete, 139
 diamentowy, 456
 instanceof, 486
 new, 120, 138, 146
 przypisania, 69, 121
 ternarny, 184

operatory
 arytmetyczne, 64
 bitowe, 175
 logiczne, 66, 67
 logiczne warunkowe, 67
 przesunięcia bitów, 180
 relacyjne, 40, 66
 skrótowe przypisania, 70
 złożone przypisania, 70

P

pakiet, 190, 269
 AWT, 465, 477
 bookpack, 272, 273
 bookpackext, 275, 276
 java, 279
 java.awt, 504
 java.awt.event, 479, 504
 java.io, 312, 324
 java.lang, 279
 java.lang.annotation, 423
 javax, 494
 javax.swing.event, 513
 mypack, 271
 NIO, 353
 Swing, 324, 493
 pakiety Java API, 279
 panel szklany, 495
 panel warstw, 495
 panel zawartości, 495
 parametr, 33, 127
 parametr T, 430
 parametry apletów, 475
 pętla
 do-while, 97
 for, 41, 90, 92, 162
 for rozszerzona, 166
 for-each, 162, 163
 while, 96
 pętle
 bez ciała, 94
 nieskończone, 94
 zagnieżdżone, 112
 plik
 Banner.java, 471
 BookDemo.java, 272
 Bubble.java, 149
 CompFiles.java, 341
 Example.class, 31
 Example.java, 30
 ExtendThread.java, 371
 Finalize.java, 140
 GalToLit.java, 39
 GalToLitTable.java, 45

- GenQDemo.java, 450
 - Help.java, 88, 99
 - Help3.java, 109
 - HelpClassDemo.java, 130
 - ICharQ.java, 286
 - IGenQ.java, 448
 - IQDemo.java, 288
 - LogicalOpTable.java, 74
 - QDemo.java, 159
 - QDemo2.java, 207
 - QExcDemo.java, 317
 - QSDemo.java, 216
 - ShowBitsDemo.java, 182
 - Sound.java, 55
 - SwingFC.java, 516
 - TrafficLigthDemo.java, 408
 - TruckDemo.java, 242
 - UseMain.java, 396
 - VehicleDemo.class, 118
 - VehicleDemo.java, 118
 - pliki
 - .class, 270
 - HTML, 466
 - o dostępie swobodnym, 343
 - pomocy, 356
 - źródłowe, 30
 - polimorfizm, 28, 204
 - pomoc, 130, 355
 - porównywanie plików, 341, 516
 - postać binarna, 176
 - priorytet operatorów, 74
 - priorytet wątku, 378
 - proces hermetyzacji, 414
 - program, 30
 - appletviewer, 465, 522
 - CopyFile, 337
 - do zamiany liter, 176
 - javadoc, 579
 - SwingDemo, 498
 - sygnalizacji świetlnej, 408
 - szyfrujący, 178
 - wielowątkowy, 363
 - programowanie
 - obiektywne, 26
 - równoległe, 386
 - strukturalne, 26
 - przechwytywanie
 - wyjątków, 300
 - wyjątków klas pochodnych, 303
 - wyjątków klasy bazowej, 303
 - przeciążanie konstruktorów, 205
 - przeciążanie metod, 201, 204, 225
 - przekazywanie
 - argumentów, 196
 - argumentów w górę, 247
 - argumentu przez referencję, 197
 - argumentu przez wartość, 197
 - obiektów, 195
 - przenośność, 23, 25
 - przerwanie programu, Ctrl+C, 391
 - przerywanie pętli, 102
 - przesłanianie metod, 252, 255
 - przesłanianie metod w klasie TwoDShape, 257
 - przesłanianie nazw, 63
 - przestrzeni nazw, 270
 - przypisanie, 120
 - przypisywanie referencji tablic, 155
 - pusta instrukcja, 94
- ## R
- referencje, 120
 - interfejsu, 284
 - klasy bazowej, 248
 - obiektów klas pochodnych, 250
 - tablicy, 155
 - rekurencja, 210
 - ręczne opakowywanie i wypakowywanie, 415
 - rozproszone, 25
 - rozszerzenie .class, 31
 - rozszerzenie .java, 30
 - rzutowanie, 442
 - rzutowanie typów, 72
- ## S
- schemat klasy Triangle, 231
 - schemat tablicy, 147
 - sekwencje specjalne, 57
 - sekwencje znaków, 57
 - selektor, 353
 - separator, 44
 - serwlet, 25
 - składnia (...), 226
 - składowa length, 156
 - składowa protected, 275, 278
 - składowa prywatna, 192
 - składowe klasy, 27
 - słowo kluczowe, 47
 - assert, 487
 - class, 116
 - enum, 403
 - extends, 230, 291
 - false, 47
 - final, 264
 - finally, 309
 - import, 420
 - instanceof, 487
 - interface, 280, 423

słowo kluczowe
 native, 488
 null, 47
 public, 33
 static, 33, 212, 420
 strictfp, 487
 super, 237, 240, 253
 synchronized, 381
 this, 142, 240, 489
 transient, 486
 true, 47
 try i catch, 297
 void, 33
 volatile, 486

słuchacz, 480
 specyfikacja zasobu, 336
 stała final, 266
 stałe własnego typu, 400
 stałe współdzielone, 291
 stałe wyliczeniowe, 400, 401
 sterowanie cyklem wykonania apletu, 468
 sterowanie instrukcją switch, 172
 stos, 28, 158
 strumień, 324
 bajtowy, 324, 345
 błędów, 327
 InputStream, 339
 InputStreamReader, 345
 predefiniowany, 327
 wejścia, 324, 328
 wyjścia, 324, 327
 znakowy, 324, 345
 znakowy FileReader, 352
 znakowy FileWriter, 351
 znakowy PrintWriter, 349

sygnatura, 205
 synchronizacja, 364, 380
 dostępu do metody, 381
 dostępu do obiektu, 381
 instrukcji, 384
 metod, 381, 384

system ósemkowy, 57
 system pomocy, 355
 system szesnastkowy, 57
 szkielet apletu, 468

Ś

ścieżka dostępu do katalogu, 271
 średnik, 44

T

tabela prawdy, 74
 tablice, 145
 dwuwymiarowe, 151

jednowymiarowe, 146
 łańcuchów, 170
 nieregularne, 152
 tablic, 165
 typu T, 460
 wielowymiarowe, 153

technologia HotSpot, 24
 terminologia Javy, 25
 tryb SINGLE_SELECTI, 513

tworzenie
 adnotacji, 423
 aplikacji Swing, 497, 522
 instancji T, 458
 łańcuchów, 167
 obiektu, 117
 przycisku, 503
 wątku, 365
 wielu wątków, 372

typ
 interfejsu generycznego, 447
 logiczny, 54
 surowy, 452
 surowy klasy generycznej, 453
 wyliczeniowy, 404
 zmiennej, 36
 zmiennej referencyjnej, 250

typy
 całkowite, 51
 danych, 37, 50
 generyczne, 413, 432
 ogólne, 267
 opakowujące, 414
 podstawowe, 139
 zmiennoprzecinkowe, 52

U

Unicode, 53
 ustawianie bitu, 176

W

wartość null, 348
 wartość porządkowa, 407
 wartość zmiennych, 62
 wątek, 363
 główny, 365, 369, 396
 gotowy do wykonywania, 364
 kończący działanie, 376, 392
 potomny, 375
 priorytet, 378
 rozdziału zdarzeń, 501
 wstrzymujący działanie, 392
 wykonywany, 364
 wznowiający działanie, 392

- zablokowany, 364
- zakończony, 364
- zawieszony, 364
- wczytywanie łańcuchów, 348
- widok, 493
- wielowątkowość, 25, 364, 396
- wielozadaniowość, 364
- wiersz wywołania, 173
- wnioskowanie typów, 456
- wskaźnik, 26
- wskaźnik zawartości pliku, 343
- wycieki pamięci, 331
- wyjątek, 295
 - ArithmeticException, 301, 313
 - ArrayIndexOutOfBoundsException, 149, 298, 300, 304, 313
 - AssertionError, 487
 - FileNotFoundException, 334
 - IOException, 311, 328
- wyjątki
 - niesprawdzane, 314
 - sprawdzane, 314
 - w klasie Queue, 317
 - wbudowane w Javę, 313
- wyliczanie wartości, 220
- wyliczenie, 400
- wymazywanie, 457
- wyrażenia, 75
- wyścig, 392
- wywołanie
 - konstruktora klasy bazowej, 237
 - super.show(), 254
 - System.in.read(), 80
 - this(), 489
 - konstruktorów, 247

Z

- zagnieżdżanie
 - bloków, 62
 - bloków try, 304
 - instrukcji if, 82
 - instrukcji switch, 88
 - zasięgów, 61

- zakleszczenie, 392
- zamknięcie pliku, 332
- zamykanie automatyczne pliku, 336
- zapis danych, 329
- zapis w pliku, 334
- zapis znaków w pliku, 350
- zapobieganie dziedziczeniu, 265
- zapobieganie przesłanianiu, 264
- zasięg deklaracji, 95
- zasięg deklaracji zmiennych, 60
- zbiór znaków, 353
- zdarzenia, 479
 - klasy, 480
 - model delegacji, 481
 - słuchacze, 480
 - źródła, 479
- zdarzenie ListSelectionEvent, 513, 514
- zintegrowane środowisko programisty, 29
- zmienna, 27, 35, 59
 - final, 406
 - iteracyjna, 162, 165
 - minivan, 120
 - referencyjna, 248, 285
 - sterująca, 93
 - środowiskowa CLASSPATH, 271
- zmiennie interfejsu, 280
- znacznik APPLET, 465
- znacznik OBJECT, 466
- znaczniki javadoc, 573
- znak +, 36
- znak komentarza, 277
- znaki, 53
- zwracanie obiektów, 199
- zwracanie wartości, 125

Ź

- źródła zdarzeń, 479

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA WYDAWNICZA

 **Helion SA**

Java

Wydanie V

Przewodnik dla początkujących

Java to język przeznaczony do realizacji najtrudniejszych programistycznych zadań. Świetnie sprawdza się tam, gdzie wymagane są najwyższa wydajność, niezawodność i bezpieczeństwo. Dzięki jasno sprecyzowanym zasadom, przejrzystej składni i silnemu typowaniu jest językiem łatwym w nauce i odpornym na błędy początkujących programistów. Java ma jeszcze jedną zaletę – programiści znający ten język są poszukiwani na rynku pracy w każdej liczbie. Chcesz dołączyć do tego grona?

To nie jest trudne! Na początek wystarczy Ci ta książka. Dzięki niej wejdiesz w świat Javy, klas, obiektów, polimorfizmu... Poznasz zalety i wady tego języka oraz to, jak wypada on na tle konkurencyjnych rozwiązań. Następnie zaznajomisz się ze składnią, typami danych, strukturą programu oraz mechanizmem wyjątków. Kolejne rozdziały zawierają bezcenne informacje na temat tworzenia interfejsu użytkownika, dokumentowania kodu i radzenia sobie z typowymi problemami. Ta książka gwarantuje bezbolesne rozpoczęcie przygody z językiem Java!

Zaprojektowana do łatwej nauki!

- ▶ Kluczowe umiejętności i koncepcje przedstawione na początku każdego rozdziału
- ▶ Ekspert odpowiada, czyli pytania i odpowiedzi zawierające dodatkowe informacje oraz pomocne wskazówki
- ▶ Przykłady i projekty, czyli ćwiczenia demonstrujące, jak wykorzystać w praktyce zdobyte umiejętności
- ▶ Testy sprawdzające, czyli pytania i zadania na końcu każdego rozdziału, pozwalające sprawdzić wiedzę
- ▶ Składnia ze wskazówkami, czyli przykłady kodu z komentarzami opisującymi wykorzystywane techniki programowania



Najlepszy podręcznik dla początkującego programisty!

helion.pl
księgarnia
internetowa

Nr katalogowy: 8564



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



Zamówienia telefoniczne:
0 801 339900



0 601 339900



Helion

Sprawdź najnowsze promocje:

▶ <http://helion.pl/promocje>

Książki najchętniej czytane:

▶ <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

▶ <http://helion.pl/nowosci>

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel.: 32 230 98 63

e-mail: helion@helion.pl

<http://helion.pl>



ISBN 978-83-246-3919-9



Cena 79,00 zł

Informatyka w najlepszym wydaniu

9 788324 639199