helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Java 2. Techniki zaawansowane. Wydanie II

Autorzy: Cay Horstmann, Gary Cornell
Tłumaczenie: Jaromir Senczyk
ISBN: 83-7361-842-2
Tytuł oryginału: Core Java(TM) 2, Volume II – Advanced Features (7th Edition)
Format: B5, stron: 1144
Przykłady na ftp: 1644 kB

Kompletne i niezastąpione źródło wiedzy dla doświadczonych programistów

- Kolejne wydanie doskonałego przewodnika po zaawansowanych możliwościach Javy
- Wszystkie kody źródłowe w książce zaktualizowane do J2SE 5.0
- Szczegółowe omówienie wielowątkowości, kolekcji, programowania aplikacji sieciowych i bazodanowych, bezpieczeństwa i internacjonalizacji aplikacji, obiektów rozproszonych i języka XML

Kolejne już wydanie przewodnika opisującego zaawansowane i nieznane właściwości języka Java, tym razem w wersji 5.0, to kompendium wiedzy dla wszystkich programistów zamierzających tworzyć rozbudowane aplikacje. Nowa wersja języka Java to nie tylko nowy numer – to przede wszystkim ogromna ilość nowych funkcji i możliwości, klas i obiektów. W JDK 5.0 wprowadzono nowe mechanizmy obsługi wątków i kolekcji, rozszerzono możliwości biblioteki Swing i klas wykorzystywanych do tworzenia aplikacji bazodanowych i sieciowych. „Java 2. Techniki zaawansowane. Wydanie II" przedstawia i opisuje wszystkie te nowości.

Wszystkie przykładowe programy zostały zaktualizowane do najnowszej wersji Javy i przedstawiają praktyczne rozwiązania rzeczywistych problemów, z jakimi może spotkać się twórca aplikacji w języku Java. Książka zawiera wiele nowych podrozdziałów poświęconych nowościom wprowadzonym w J2SE 5.0. Dokładnie i na przykładach opisuje zagadnienia związane z wielowątkowością, kolekcjami, metadanymi, stosowaniem języka XML, komunikacją z bazami danych i wieloma innymi elementami zaawansowanego programowania w Javie.

- Aplikacje wielowątkowe
- Kolekcje i operacje na nich
- Połączenia sieciowe
- Interfejs JDBC i LDAP
- Aplikacje rozproszone
- Technologia CORBA
- Zaawansowane możliwości bibliotek Swing i AWT
- Technologia JavaBeans
- Bezpieczeństwo aplikacji
- Internacjonalizacja
- Korzystanie z języka XML

Jeśli zamierzasz wykorzystać Javę w złożonym projekcie informatycznym, ta książka będzie dla Ciebie niezastąpiona.

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

Spis treści

Podziękowania 11

Przedmowa 13

Do Czytelnika 13
O książce 13

Rozdział 1. Wielowątkowość 17

Czym są wątki? 18
Zastosowanie wątków 23
Przerywanie wątków 29
Stany wątków 32
Nowe wątki 32
Wątki wykonywalne 33
Wątki zablokowane 33
Wątki martwe 35
Właściwości wątków 36
Priorytety wątków 36
Wątki-demony 37
Grupy wątków 37
Procedury obsługi wyjątków 39
Synchronizacja 40
Przykład wyścigu 41
Wytłumaczenie wyścigu 45
Blokady 46
Warunki 49
Słowo kluczowe synchronized 54
Bloki synchronizowane 60
Pola volatile 61
Zakleszczenia 63
Równorzędne traktowanie wątków 65
Testowanie blokad i limity czasu 65
Blokady odczytu i zapisu 67
Dlaczego metody stop i suspend nie są zalecane? 68
Kolejki blokujące 70
Kolekcje 76
Efektywne implementacje kolejki i tablicy mieszającej 76
Kolekcje CopyOnWriteArrayList i CopyOnWriteArraySet 78
Starsze kolekcje i wielowątkowość 78

Interfejsy Callable i Future ... 78
Egzekutory ... 83
Pule wątków ... 83
Wykonywanie zadań według planu ... 87
Sterowanie grupami wątków ... 88
Synchronizatory ... 89
Bariery ... 90
Rejestry odliczające ... 91
Przekaźniki ... 91
Kolejki synchroniczne ... 91
Semafory ... 92
Wątki i Swing ... 98
Zasada pojedynczego wątku ... 99
Wątek roboczy i Swing ... 104

Rozdział 2. Kolekcje ... 111

Interfejsy kolekcji ... 111
Rozdzielenie interfejsów kolekcji od ich implementacji ... 112
Interfejsy Collection i Iterator w bibliotekach języka Java ... 114
Kolekcje konkretne ... 119
Listy powiązane ... 120
Klasa ArrayList ... 128
Zbiory z kodowaniem mieszającym ... 129
Zbiory drzewiaste ... 132
Kolejki z priorytetami ... 138
Mapy ... 139
Specjalizowane klasy zbiorów i map ... 144
Szkielet kolekcji ... 149
Widoki i opakowania ... 152
Operacje masowe ... 158
Wykorzystanie biblioteki kolekcji z tradycyjnymi bibliotekami ... 159
Rozbudowywanie szkieletu ... 160
Algorytmy ... 163
Sortowanie i tasowanie ... 164
Wyszukiwanie binarne ... 167
Proste algorytmy ... 168
Programowanie własnych algorytmów ... 169
Tradycyjne kolekcje ... 171
Klasa Hashtable ... 171
Wyliczenia ... 171
Zbiory właściwości ... 172
Stosy ... 173
Zbiory bitów ... 173

Rozdział 3. Programowanie aplikacji sieciowych ... 179

Połączenia z serwerem ... 179
Implementacja serwerów ... 183
Obsługa wielu klientów ... 186
Wysyłanie poczty elektronicznej ... 189
Połączenia wykorzystujące URL ... 193
URL i URI ... 194
Zastosowanie klasy URLConnection do pobierania informacji ... 196
Wysyłanie danych do formularzy ... 205

Zaawansowane programowanie przy użyciu gniazdek sieciowych ... 214
Limity czasu gniazdek ... 214
Przerywanie operacji gniazdek ... 215
Połączenia częściowo zamknięte ... 219
Adresy internetowe ... 220

Rozdział 4. Połączenia do baz danych: JDBC ... 225

Architektura JDBC ... 226
Typy sterowników JDBC ... 227
Typowe zastosowania JDBC ... 228
Język SQL ... 230
Instalacja JDBC ... 235
Podstawowe koncepcje programowania przy użyciu JDBC ... 235
Adresy URL baz danych ... 236
Nawiązywanie połączenia ... 236
Wykonywanie poleceń języka SQL ... 241
Zaawansowane typy języka SQL ... 242
Zarządzanie połączeniami, poleceniami i zbiorami wyników ... 245
Wypełnianie bazy danych ... 246
Wykonywanie zapytań ... 249
Polecenia przygotowane ... 250
Przewijalne i aktualizowalne zbiory wyników zapytań ... 258
Przewijalne zbiory rekordów ... 259
Aktualizowalne zbiory rekordów ... 262
Metadane ... 266
Zbiory rekordów ... 275
Buforowane zbiory rekordów ... 276
Transakcje ... 285
Punkty kontrolne ... 286
Aktualizacje wsadowe (JDBC 2) ... 286
Zaawansowane zarządzanie połączeniami ... 289
Wprowadzenie do LDAP ... 290
Konfiguracja serwera LDAP ... 291
Dostęp do informacji katalogu LDAP ... 294

Rozdział 5. Obiekty rozproszone ... 305

Role klienta i serwera ... 306
Wywołania zdalnych metod ... 308
Namiastka i szeregowanie parametrów ... 309
Dynamiczne ładowanie klas ... 311
Konfiguracja wywołania zdalnych metod ... 312
Interfejsy i implementacje ... 312
Generowanie klasy namiastki ... 315
Odnajdywanie obiektów serwera ... 315
Po stronie klienta ... 319
Przygotowanie wdrożenia ... 324
Wdrożenie programu ... 326
Przekazywanie parametrów zdalnym metodom ... 329
Przekazywanie lokalnych obiektów ... 329
Przekazywanie zdalnych obiektów ... 341
Zdalne obiekty i metody equals oraz hashCode ... 343
Klonowanie zdalnych obiektów ... 344

Aktywacja obiektów serwera ... 344
Java IDL i CORBA ... 350
Język IDL ... 351
Przykład aplikacji CORBA ... 356
Implementacja serwerów CORBA ... 365
Wywołania zdalnych metod i SOAP ... 370

Rozdział 6. Zaawansowane możliwości pakietu Swing ... 377

Listy ... 377
Komponent JList ... 378
Modele list ... 384
Wstawianie i usuwanie ... 388
Odrysowywanie zawartości listy ... 390
Drzewa ... 395
Najprostsze drzewa ... 396
Przeglądanie węzłów ... 411
Rysowanie węzłów ... 412
Nasłuchiwanie zdarzeń w drzewach ... 419
Własne modele drzew ... 425
Tabele ... 433
Najprostsze tabele ... 433
Modele tabel ... 436
Filtry sortujące ... 445
Rysowanie i edytowanie zawartości komórek ... 451
Operacje na wierszach i kolumnach ... 464
Wybór wierszy, kolumn i komórek ... 465
Komponenty formatujące tekst ... 473
Wskaźniki postępu ... 479
Paski postępu ... 480
Monitory postępu ... 484
Monitorowanie postępu strumieni wejścia ... 489
Organizatory komponentów ... 494
Panele dzielone ... 495
Panele z zakładkami ... 499
Panele pulpitu i ramki wewnętrzne ... 504
Rozmieszczenie kaskadowe i sąsiadujące ... 507
Zgłaszanie weta do zmiany właściwości ... 510

Rozdział 7. Zaawansowane możliwości biblioteki AWT ... 521

Potokowe tworzenie grafiki ... 522
Figury ... 524
Wykorzystanie klas obiektów graficznych ... 526
Pola ... 539
Ślad pędzla ... 543
Wypełnienia ... 550
Przekształcenia układu współrzędnych ... 556
Przycinanie ... 565
Przezroczystość i składanie obrazów ... 569
Wskazówki operacji graficznych ... 577
Czytanie i zapisywanie plików graficznych ... 583
Wykorzystanie obiektów zapisu i odczytu plików graficznych ... 583
Odczyt i zapis plików zawierających sekwencje obrazów ... 585

Operacje na obrazach ... 595
Dostęp do danych obrazu ... 595
Filtrowanie obrazów ... 602
Drukowanie ... 610
Drukowanie grafiki ... 611
Drukowanie wielu stron ... 621
Podgląd wydruku ... 623
Usługi drukowania ... 631
Usługi drukowania za pośrednictwem strumieni ... 637
Atrybuty drukowania ... 642
Schowek ... 649
Klasy i interfejsy umożliwiające przekazywanie danych ... 650
Przekazywanie tekstu ... 651
Interfejs Transferable i formaty danych ... 655
Przekazywanie obrazów za pomocą schowka ... 657
Wykorzystanie lokalnego schowka do przekazywania referencji obiektów ... 662
Wykorzystanie schowka systemowego do przekazywania obiektów Java ... 668
Mechanizm „przeciągnij i upuść” ... 672
Cele mechanizmu „przeciągnij i upuść” ... 674
Źródła mechanizmu „przeciągnij i upuść” ... 683
Przekazywanie danych pomiędzy komponentami Swing ... 688

Rozdział 8. JavaBeans ... 693

Dlaczego ziarnka? ... 694
Proces tworzenia ziarnek JavaBeans ... 695
Wykorzystanie ziarnek do tworzenia aplikacji ... 698
Umieszczanie ziarnek w plikach JAR ... 699
Korzystanie z ziarnek ... 700
Wzorce nazw właściwości ziarnek i zdarzeń ... 705
Typy właściwości ziarnek ... 708
Właściwości proste ... 708
Właściwości indeksowane ... 709
Właściwości powiązane ... 710
Właściwości ograniczone ... 712
Klasa informacyjna ziarnka ... 718
Edytory właściwości ... 723
Implementacja edytora właściwości ... 730
Indywidualizacja ziarnka ... 744
Implementacja klasy indywidualizacji ... 746
Trwałość ziarnek JavaBeans ... 753
Zastosowanie mechanizmu trwałości JavaBeans dla dowolnych danych ... 758
Kompletny przykład zastosowania trwałości JavaBeans ... 764

Rozdział 9. Bezpieczeństwo ... 775

Ładowanie klas ... 776
Implementacja własnej procedury ładującej ... 779
Weryfikacja kodu maszyny wirtualnej ... 784
Menedżery bezpieczeństwa i pozwolenia ... 789
Bezpieczeństwo na platformie Java 2 ... 791
Pliki polityki bezpieczeństwa ... 794
Tworzenie własnych klas pozwoleń ... 801
Implementacja klasy pozwoleń ... 802

Tworzenie własnych menedżerów bezpieczeństwa ... 808
Uwierzytelnianie użytkowników ... 815
Moduły JAAS ... 820
Podpis cyfrowy ... 829
Skróty wiadomości ... 830
Podpisywanie wiadomości ... 835
Uwierzytelnianie wiadomości ... 843
Certyfikaty X.509 ... 845
Tworzenie certyfikatów ... 847
Podpisywanie certyfikatów ... 849
Podpisywanie kodu ... 857
Podpisywanie plików JAR ... 857
Certyfikaty twórców oprogramowania ... 861
Szyfrowanie ... 863
Szyfrowanie symetryczne ... 863
Strumienie szyfrujące ... 870
Szyfrowanie kluczem publicznym ... 871

Rozdział 10. Internacjonalizacja ... 877

Lokalizatory ... 878
Formaty liczby ... 883
Waluty ... 889
Data i czas ... 890
Porządek alfabetyczny ... 897
Formatowanie komunikatów ... 905
Formatowanie z wariantami ... 907
Pliki tekstowe i zbiory znaków ... 909
Internacjonalizacja a pliki źródłowe programów ... 909
Komplety zasobów ... 910
Lokalizacja zasobów ... 911
Pliki właściwości ... 912
Klasy kompletów zasobów ... 913
Kompletny przykład ... 915

Rozdział 11. Metody macierzyste ... 929

Wywołania funkcji języka C z programów w języku Java ... 931
Wykorzystanie funkcji printf ... 932
Numeryczne parametry metod i wartości zwracane ... 937
Wykorzystanie funkcji printf do formatowania liczb ... 937
Łańcuchy znaków jako parametry ... 938
Wywołanie funkcji sprintf przez metodę macierzystą ... 942
Dostęp do składowych obiektu ... 944
Dostęp do pól instancji ... 944
Dostęp do pól statycznych ... 949
Sygnatury ... 949
Wywoływanie metod języka Java ... 951
Wywoływanie metod obiektów ... 951
Wywoływanie metod statycznych ... 952
Konstruktory ... 953
Alternatywne sposoby wywoływania metod ... 954
Tablice ... 958
Obsługa błędów ... 963

Interfejs programowy wywołań języka Java ... 967
Kompletny przykład: dostęp do rejestru systemu Windows ... 971
Rejestr systemu Windows ... 971
Interfejs dostępu do rejestru na platformie Java ... 972
Implementacja dostępu do rejestru za pomocą metod macierzystych ... 973

Rozdział 12. Język XML ... 987

Wprowadzenie do języka XML ... 988
Struktura dokumentu XML ... 990
Parsowanie dokumentów XML ... 993
Kontrola poprawności dokumentów XML ... 1003
Definicje typów dokumentów ... 1005
XML Schema ... 1012
Praktyczny przykład ... 1014
Wyszukiwanie infomacji i XPath ... 1028
Przestrzenie nazw ... 1033
Wykorzystanie parsera SAX ... 1036
Tworzenie dokumentów XML ... 1041
Przekształcenia XSL ... 1049

Rozdział 13. Adnotacje ... 1059

Umieszczanie metadanych w programach ... 1060
Przykład — adnotacje obsługi zdarzeń ... 1061
Składnia adnotacji ... 1066
Adnotacje standardowe ... 1070
Adnotacje regularne ... 1070
Metaadnotacje ... 1071
Narzędzie apt do przetwarzania adnotacji w kodzie źródłowym ... 1074
Inżynieria kodu bajtowego ... 1080
Modyfikacja kodu bajtowego podczas ładowania ... 1089

Skorowidz ... 1093

1

Wielowątkowość

W tym rozdziale:

- Czym są wątki?
- Przerywanie wątków.
- Stany wątków.
- Właściwości wątków.
- Synchronizacja.
- Kolejki blokujące.
- Kolekcje.
- Interfejsy Callable i Future.
- Egzekutory.
- Synchronizatory.
- Wątki i Swing.

Czytelnik z pewnością wie, że *wielozadaniowość* oznacza możliwość pracy wielu programów równocześnie. Dzięki wielozadaniowości możemy w czasie edycji dokumentu drukować inny dokument bądź wysyłać faks. Oczywiście wtedy, gdy mamy do dyspozycji tylko maszynę o pojedynczym procesorze, uzyskujemy jedynie wrażenie równoczesnego wykonywania wielu programów, ponieważ system operacyjny przydziela czas procesora kolejnym zadaniom. Takie zarządzanie przydziałem procesora jest tym bardziej możliwe, że wiele zadań absorbuje jego moc obliczeniową w znikomym stopniu.

Wielozadaniowość jest realizowana na dwa sposoby: z *wywłaszczaniem* i bez. W pierwszym przypadku system operacyjny samodzielnie podejmuje decyzję o przydziale procesora kolejnym zadaniom, natomiast w drugim wykonywanie zadania może zostać przerwane tylko wtedy, jeśli zgodzi się ono oddać sterowanie. Starsze systemy operacyjne, takie jak Windows 3.1 i Mac OS 9, pracują na zasadzie wielozadaniowości bez wywłaszczania. Wten sam sposób działają też systemy operacyjne prostych urządzeń, na przykład telefonów komórkowych. Natomiast system UNIX i jego pochodne, a także systemy Windows NT/XP (oraz Windows 9x w przypadku aplikacji 32-bitowych) i OS X stosują wielozadaniowość

z wywłaszczaniem. Chociaż realizacja wielozadaniowości z wywłaszczaniem jest dużo trudniejsza, to rozwiązanie takie jest bardziej efektywne, ponieważ w przypadku wielozadaniowości bez wywłaszczania niewłaściwie zachowująca się aplikacja może wstrzymać wykonywanie pozostałych zadań w systemie.

Wielowątkowość rozszerza ideę wielozadaniowości w ten sposób, że każdy z programów może wykonywać równocześnie wiele zadań. Zadania te nazywamy *wątkami*. Program, który wykonuje więcej niż jeden wątek, nazywamy *wielowątkowym*.

Zasadnicza różnica pomiędzy zadaniami a wątkami polega na tym, że podczas gdy każde zadanie dysponuje własnym oddzielnym zestawem danych (zmiennych), to wątki operują na wspólnych danych. Utworzenie bądź usunięcie wątku wiąże się z dużo mniejszym nakładem ze strony systemu operacyjnego niż w przypadku tych samych operacji dla zadania. Podobnie komunikacja między zadaniami jest mniej efektywna niż pomiędzy wątkami. Dlatego też zdecydowana większość współczesnych systemów operacyjnych obsługuje wielowątkowość.

Wielowątkowość okazuje się niesłychanie przydatna w praktyce. Przeglądarka internetowa, wykorzystując wątki, pozwala jednocześnie załadować wiele obrazów na stronie. Klient poczty elektronicznej umożliwia czytania poczty, w trakcie pobierając nowe wiadomości. Także Java używa dodatkowego wątku, aby odzyskać w tle niewykorzystywaną przez program pamięć. Pogramy wyposażone w graficzny interfejs użytkownika stosują osobny wątek do uzyskiwania informacji o zdarzeniach zachodzących w systemie okienkowym. W rozdziale tym pokażemy, w jaki sposób wykorzystać zalety wielowątkowości w aplikacjach tworzonych w języku Java.

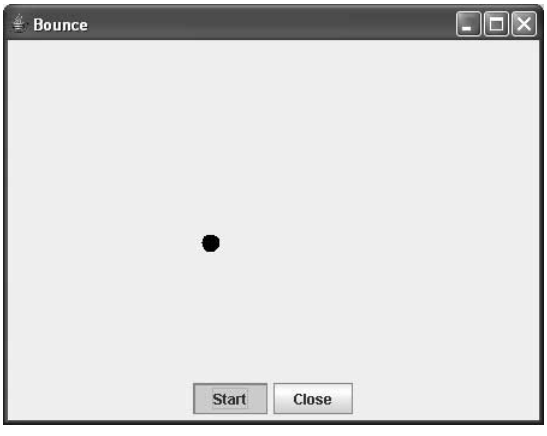
Pakiet JDK 5.0 udostępnia wiele nowych klas i interfejsów dostarczających zaawansowanej implementacji mechanizmów wielowątkowych. W rozdziale tym omówimy nowe możliwości JDK 5.0, a także klasyczne mechanizmy synchronizacji i pomożemy Ci dokonać właściwego wyboru pomiędzy nimi.

Wielowątkowość nie jest prostym zagadnieniem. W rozdziale tym przedstawiamy wszystkie narzędzia, które udostępnia język Java do programowania wątków. Wyjaśniamy przy tym sposób ich wykorzystania, związane z tym ograniczenia i ilustrujemy całość prostymi, ale typowymi przykładami. W bardziej złożonych przypadkach Czytelnik powinien jednak skorzystać z bardziej specjalistycznej literatury, na przykład *Concurrent Programming in Java* autorstwa Douga Lea (Addison-Wesley, 1999).

Czym są wątki?

Zapoznajmy się najpierw z przykładowym programem, który nie używa wątków i w konsekwencji użytkownik nie może wykonać za jego pomocą wielu działań naraz. Później pokażemy, jak łatwo usunąć te niedogodności, wprowadzając do programu wątki. Program ten będzie animacją piłki odbijającej się od ramki okienka (patrz rysunek 1.1).

Wybranie przez użytkownika przycisku *Start* powoduje pojawienie się w lewym górnym rogu okienka piłki, która rozpoczyna swój ruch. Metoda obsługi zdarzenia dla przycisku *Start* wywołuje metodę `addBall`. Metoda ta zawiera pętlę, w której 1000 razy wykonywana jest metoda `move` powodująca niewielkie przesunięcie rysunku piłki, ewentualną zmianę kierunku ruchu w przypadku odbicia od ramki i odrysowanie tła okna.



Rysunek 1.1.
Animacja piłki odbijającej się od ramki okienka

```
Ball ball = new Ball();
panel.add(ball);

for (int i = 1; i <= STEPS; i++)
{
  ball.move(panel.getBounds());
  panel.paint(panel.getGraphics());
  Thread.sleep(DELAY);
}
```

Metoda statyczna `sleep` klasy `Thread` powoduje zawieszenie wykonywania pętli na określoną liczbę milisekund.

Wywołanie metody `Thread.sleep` nie tworzy nowego wątku — metoda `sleep` jest statyczną metodą klasy `Thread`, która powoduje wstrzymanie wykonywania aktywnego wątku na określony czas.

Metoda `sleep` może spowodować wystąpienie wyjątku `InterruptedException`. Wyjątek ten oraz sposób jego obsługi omówimy dokładniej w dalszej części rozdziału. W obecnej wersji programu jego wystąpienie spowoduje jedynie przerwanie animacji.

Animacja ruchu piłki w obecnej wersji programu absorbuje całkowicie jedyny wątek aplikacji w jej obecnej postaci. Jeśli użytkownik zechce, zanim pętla wykona się 1000 razy, przerwać działanie programu, wybierając przycisk `Close`, to nie przyniesie to zamierzonego efektu. Program nie może bowiem obsłużyć tego zdarzenia, dopóki nie zakończy animacji.

Przyglądając się pełnemu tekstowi programu zamieszczonemu poniżej, zauważymy wywołanie

```
canvas.paint(canvas.getGraphics())
```

wewnątrz metody `move` klasy `Ball`. Rozwiązanie to może wydawać się dziwne, ponieważ zwykle wywołujemy metodę `repaint` i pozwalamy AWT zająć się określeniem kontekstu graficznego i odrysowaniem zawartości okna. Jednak jeśli postąpimy w ten sposób także w tym programie, to odrysowanie nie będzie mogło się odbyć, ponieważ metoda `addBall` zmonopolizowała przetwarzanie. W następnej wersji programu, która będzie obliczać kolejne pozycji piłki w osobnym wątku, z powrotem będziemy mogli zastosować wywołanie metody `repaint`.

Zachowanie obecnej wersji programu jest dalekie od doskonałości. W większości przypadków gdy program wykonuje czasochłonne operacje, musi istnieć możliwość ich przerwania. Typowym przykładem są wszelkie programy czytające dane z sieci. Zawsze powinna istnieć możliwość przerwania na przykład procesu ładowania dużego obrazka. Jeśli po obejrzeniu jego fragmentu użytkownik stwierdzi, że nie interesuje go całość, to wybranie przycisku *Stop* lub *Back* powinno spowodować przerwanie procesu ładowania. W dalszej części rozdziału pokażemy, w jaki sposób można zapewnić użytkownikowi pełną kontrolę nad działaniem programu przez wykonywanie jego kluczowych fragmentów w osobnym *wątku*.

Listing 1.1 zawiera pełen tekst źródłowy obecnej wersji programu.

Listing 1.1. *Bounce.java*

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;

/**
   Animacja odbijającej się piłki.
*/
public class Bounce
{
   public static void main(String[] args)
   {
      JFrame frame = new BounceFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Piłka poruszająca się i odbijająca od krawędzi
   komponentu.
*/
class Ball
{
   /**
      Przesuwa piłkę do następnej pozycji, zmieniając kierunek ruchu,
      jeśli natrafi na krawędź.
   */
   public void move(Rectangle2D bounds)
   {
      x += dx;
      y += dy;
      if (x < bounds.getMinX())
      {
         x = bounds.getMinX();
         dx = -dx;
      }
      if (x + XSIZE >= bounds.getMaxX())
      {
         x = bounds.getMaxX() - XSIZE;
         dx = -dx;
      }
```

```
      if (y < bounds.getMinY())
      {
         y = bounds.getMinY();
         dy = -dy;
      }
      if (y + YSIZE >= bounds.getMaxY())
      {
         y = bounds.getMaxY() - YSIZE;
         dy = -dy;
      }
   }

   /**
      Tworzy kształt piłki dla bieżącej pozycji.
   */
   public Ellipse2D getShape()
   {
      return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
   }

   private static final int XSIZE = 15;
   private static final int YSIZE = 15;
   private double x = 0;
   private double y = 0;
   private double dx = 1;
   private double dy = 1;
}

/**
   Panel rysujący piłki.
*/
class BallPanel extends JPanel
{
   /**
      Dodaje piłkę do panelu.
      @param b dodawana piłka
   */
   public void add(Ball b)
   {
      balls.add(b);
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;
      for (Ball b : balls)
      {
         g2.fill(b.getShape());
      }
   }

   private ArrayList<Ball> balls = new ArrayList<Ball>();
}

/**
      Ramka zawierająca tło i przyciski.
```

```
*/
class BounceFrame extends JFrame
{
   /**
      Konstruuje ramkę zawierającą panel, w której animowana będzie piłka
      i pokazane przyciski Start i Close

   */
   public BounceFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      setTitle("Bounce");

      panel = new BallPanel();
      add(panel, BorderLayout.CENTER);
      JPanel buttonPanel = new JPanel();
      addButton(buttonPanel, "Start",
         new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               addBall();
            }
         });

      addButton(buttonPanel, "Close",
         new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
      add(buttonPanel, BorderLayout.SOUTH);
   }

   /**
      Dodaje przycisk do kontenera.
      @param c kontener
      @param title nazwa przycisku
      @param listener obiekt nasłuchujący przycisku

   */
   public void addButton(Container c, String title, ActionListener listener)
   {
      JButton button = new JButton(title);
      c.add(button);
      button.addActionListener(listener);
   }

   /**
      Dodaje piłkę do panelu
Rysuje piłkę i animuje ją 1000 razy
   */
   public void addBall()
   {
      try
```

```
      {
         Ball ball = new Ball();
         panel.add(ball);

         for (int i = 1; i <= STEPS; i++)
         {
            ball.move(panel.getBounds());
            panel.paint(panel.getGraphics());
            Thread.sleep(DELAY);
         }
      }
      catch (InterruptedException e)
      {
      }
   }

   private BallPanel panel;
   public static final int DEFAULT_WIDTH = 450;
   public static final int DEFAULT_HEIGHT = 350;
   public static final int STEPS = 1000;
   public static final int DELAY = 3;
}
```

`java.lang.Thread 1.0`

- `static void sleep(long millis)` zawiesza wykonanie wątku na określoną liczbę milisekund.

  Parametry: `millis` liczba milisekund zawieszenia wątku

Zastosowanie wątków

Program animacji piłki może lepiej odpowiadać na akcje użytkownika, jeśli kod odpowiedzialny za animację umieścimy w osobnym wątku. Rozwiązanie takie pozwoli nam nawet animować wiele piłek. Animacja każdej z nich odbywać będzie się w osobnym wątku. Równolegle z wątkami animacji działać będzie również *wątek obsługi zdarzeń* AWT zajmujący się obsługą zdarzeń związanych z interfejsem użytkownika. Ponieważ wszystkie wątki mają równą szansę wykonania, to główny wątek programu może teraz uzyskać infomację o wybraniu przez użytkownika przycisku *Close* i odpowiednio zareagować na to zdarzenie. A oto prosty sposób na uruchomienie zadania w osobnym wątku:

1. Kod realizujący zadanie umieszczamy wewnątrz metody `run` klasy implementującej interfejs `Runnable`. Interfejs ten jest bardzo prosty i posiada tylko jedną metodę:

   ```
   public interface Runnable
   {
      void run();
   }
   ```

   Klasę implementującą ten interfejs tworzymy w następujący sposób:

```
class MyRunnable implements Runnable
{
   public void run()
   {
      kod zadania
   }
}
```

2. Następnie tworzymy obiekt naszej klasy:

```
Runnable r = new MyRunnable();
```

3. Tworzymy obiekt `Thread` na podstawie obiektu `Runnable`:

```
Thread t = new Thread(r);
```

4. Uruchamiamy wątek:

```
t.start();
```

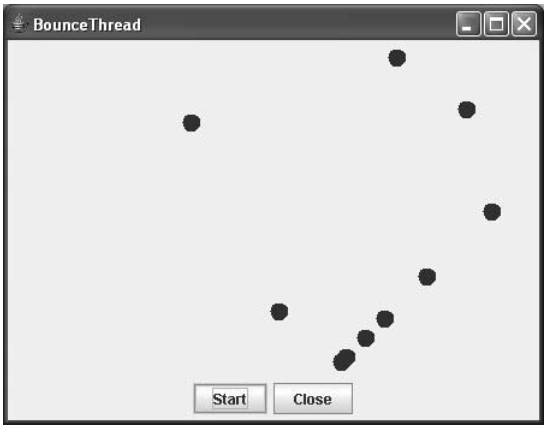
Aby kod animacji piłki wykonywany był we własnym wątku, wystarczy umieścić go wewnątrz metody `run` klasy `BallRunnable`:

```
class BallRunnable implements Runnable
{
   . . .
   public void run()
   {
      try
      {
         for (int i = 1; i <= STEPS; i++)
         {
            ball.move(component.getBounds());
            component.repaint();
            Thread.sleep(DELAY);
         }
      }
      catch (InterruptedException e)
      {
      }
   }
   . . .
}
```

Powyższy fragment kodu zawiera także obsługę wyjątku `InterruptedException`, którego wystąpienie może spowodować metoda `sleep`. Wyjątek ten omówimy w następnym punkcie. Typowo, aby zakończyć wykonywanie wątku, przerywa się jego działanie. Zgodnie z tą zasadą wystąpienie wyjątku `InterruptedException` spowoduje zakończenie wykonywania naszej metody `run`.

Za każdym razem, gdy użytkownik wybierze przycisk *Start*, metoda `addBall` uruchamia nowy wątek (patrz rysunek 1.2):

```
Ball b = new Ball();
panel.add(b);
Runnable r = new BallRunnable(b, panel);
Thread t = new Thread(r);
t.start();
```



Rysunek 1.2. *Animacja wielowątkowa*

I to wszystko! W ten sposób dowiedziałeś się, w jaki sposób uruchamiać równolegle działające wątki. Pozostała część tego rozdziału poświęcona jest interakcjom pomiędzy wątkami.

Wątek możemy również zdefiniować, tworząc klasę pochodną klasy `Thread`:

```
class MyThread extends Thread
{
   public void run()
   {
      kod zadania
   }
}
```

Następnie tworzymy obiekt tej klasy pochodnej i wywołujemy jego metodę `start`. Rozwiązanie takie nie jest jednak zalecane. Należy dążyć do oddzielenia wykonywanego *zadania* od *mechanizmu* jego wykonywania. W sytuacji, gdy musimy wykonać dużą liczbę zadań, tworzenie wątku dla każdego z nich jest zbyt kosztowne i lepiej jest wykorzystać pulę wątków (patrz strona 83.).

Nie należy wywoływać bezpośrednio metody `run` — zostanie ona wywołana przez metodę `start`, w momencie gdy wątek jest gotowy do rozpoczęcia działania. Bezpośrednie wywołanie metody `run` spowoduje jej wykonanie w bieżącym wątku zamiast utworzenia nowego.

Listing 1.2 zawiera kompletny kod animacji wielowątkowej.

Listing 1.2. *BounceThread.java*

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;
```

```
/**
   Animacja odbijającej się piłki.
*/
public class BounceThread
{
   public static void main(String[] args)
   {
      JFrame frame = new BounceFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Animacja odbijającej się piłki.
*/
class BallRunnable implements Runnable
{
   /**
      Konstruktor.
      @aBall piłka
      @aPanel komponent, w którym będzie animowana
   */
   public BallRunnable(Ball aBall, Component aComponent)
   {
      ball = aBall;
      component = aComponent;
   }

   public void run()
   {
      try
      {
         for (int i = 1; i <= STEPS; i++)
         {
            ball.move(component.getBounds());
            component.repaint();
            Thread.sleep(DELAY);
         }
      }
      catch (InterruptedException e)
      {
      }
   }

   private Ball ball;
   private Component component;
   public static final int STEPS = 1000;
   public static final int DELAY = 5;
}

/**
   Piłka poruszająca się i odbijająca od krawędzi
   komponentu.
*/
class Ball
{
```

```
   /**
      Przesuwa piłkę do następnej pozycji, zmieniając kierunek ruchu,
      jeśli natrafi na krawędź.
   */
   public void move(Rectangle2D bounds)
   {
      x += dx;
      y += dy;
      if (x < bounds.getMinX())
      {
         x = bounds.getMinX();
         dx = -dx;
      }
      if (x + XSIZE >= bounds.getMaxX())
      {
         x = bounds.getMaxX() - XSIZE;
         dx = -dx;
      }
      if (y < bounds.getMinY())
      {
         y = bounds.getMinY();
         dy = -dy;
      }
      if (y + YSIZE >= bounds.getMaxY())
      {
         y = bounds.getMaxY() - YSIZE;
         dy = -dy;
      }
   }

   /**
      Tworzy kształt piłki dla bieżącej pozycji.
   */
   public Ellipse2D getShape()
   {
      return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
   }

   private static final int XSIZE = 15;
   private static final int YSIZE = 15;
   private double x = 0;
   private double y = 0;
   private double dx = 1;
   private double dy = 1;
}

/**
   Panel rysujący piłkę.
*/
class BallPanel extends JPanel
{
   /**
      Dodaje piłkę do panelu
      @param b dodawana piłka
   */
   public void add(Ball b)
   {
      balls.add(b);
```

```
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;
      for (Ball b : balls)
      {
         g2.fill(b.getShape());
      }
   }

   private ArrayList<Ball> balls = new ArrayList<Ball>();
}

/**
   Ramka zawierająca panel i przyciski.
*/
class BounceFrame extends JFrame
{
   /**
      Konstruuje ramkę z panelem, w której animowana będzie piłka
      i umieszczone przyciski Start i Close
   */
   public BounceFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      setTitle("BounceThread");

      panel = new BallPanel();
      add(panel, BorderLayout.CENTER);
      JPanel buttonPanel = new JPanel();
      addButton(buttonPanel, "Start",
         new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               addBall();
            }
         });

      addButton(buttonPanel, "Close",
         new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
      add(buttonPanel, BorderLayout.SOUTH);
   }

   /**
      Dodaje przycisk do kontenera.
      @param c kontener
      @param title nazwa przycisku
      @param listener obiekt nasłuchujący przycisku
   */
```

```
   public void addButton(Container c, String title, ActionListener listener)
   {
      JButton button = new JButton(title);
      c.add(button);
      button.addActionListener(listener);
   }

   /**
      Tworzy piłkę i uruchamia nowy wątek jej animacji.
   */
   public void addBall()
   {
      Ball b = new Ball();
      panel.add(b);
      Runnable r = new BallRunnable(b, panel);
      Thread t = new Thread(r);
      t.start();
   }

   private BallPanel panel;
   public static final int DEFAULT_WIDTH = 450;
   public static final int DEFAULT_HEIGHT = 350;
   public static final int STEPS = 1000;
   public static final int DELAY = 3;
}
```

java.lang.Thread 1.0

- `Thread(Runnable target)` tworzy nowy wątek, który wykonywać będzie metodę run obiektu `target`.
- `void start()` uruchamia ten wątek i powoduje wywołanie metody `run`. Metoda `start` oddaje natychmiast sterowanie do metody, która ją wywołała, a nowy wątek wykonywany jest równolegle.
- `void run()` wywołuje metodę `run` odpowiedniego obiektu implementującego interfejs `Runnable`.

java.lang.Runnable 1.0

- `void run()` — tę metodę należy zastąpić własną wersją zawierającą kod, który ma być wykonywany w nowym wątku.

Przerywanie wątków

Wykonanie wątku kończy się w momencie, gdy metoda `run` zwraca sterowanie. (W pierwszej wersji języka Java istniała także metoda `stop`, za pomocą której jeden wątek mógł zakończyć wykonywanie innego. Obecnie metoda ta nie jest stosowana, a przyczyny tego omówimy w dalszej części rozdziału).

Obecnie nie istnieje więc sposób, aby *wymusić* zakończenie wątku. Można jednak *zażądać* zakończenia wątku za pomocą metody `interrupt`.

Wywołanie metody `interrupt` powoduje nadanie wątkowi *statusu przerwania*. Status ten jest reprezentowany za pomocą znacznika logicznego. Wartość tego znacznika powinna być okresowo sprawdzana przez wątek.

Aby sprawdzić status przerwania wątku, należy najpierw wywołać metodę statyczną `Thread.currentThread`, aby uzyskać bieżący wątek, a następnie wywołać metodę `isInterrupted` sprawdzającą jego status:

```
while (!Thread.currentThread().isInterrupted() && są działania do wykonania)
{
   wykonaj te działania
}
```

Jednak wątek, którego wykonywanie zostało zablokowane, nie może sprawdzić własnego statusu przerwania. I właśnie w takim przypadku pomocny jest wyjątek `InterruptedException`. Jeśli metoda `interrupt` zostanie wywołana dla wątku, którego wykonanie jest zablokowane, to blokująca go metoda `sleep` lub `wait` zostanie przerwana i pojawi się wyjątek `InterruptedException`.

Środowisko języka Java nie wymaga wcale, aby wątek przerwany w taki sposób zakończył swoje wykonywanie. Przerwanie wątku sygnalizuje jedynie pojawienie się pewnego żądania. Przerwany wątek sam decyduje, w jaki sposób zareagować na takie żądanie. Wykonanie niektórych wątków w programie może być na tyle ważne, że po prostu zignorują one wyjątek i kontynuować będą swoje działanie. Najczęściej jednak wątek zinterpretuje wyjątek jako żądanie zakończenia wykonywania. Metoda `run` takiego wątku wygląda wtedy następująco:

```
public void run()
{
   try
   {
      . . .
      while (!Thread.currentThread().isInterrupted() && są działania do wykonania)
      {
         wykonaj te działania
      }

   }
   catch(InterruptedException exception)
   {
      // wątek przerwany w trakcie metody sleep lub wait
   }
   finally
   {
      wątek „sprząta po sobie”, jeśli to konieczne
   }
   // zakończenie wykonywania metody run i tym samym wątku
}
```

Sprawdzenie statusu przerwania wątku za pomocą metody `isInterrupted` nie jest konieczne, gdy metoda `sleep` wywoływana jest po każdej iteracji przetwarzania. Gdy wątek posiada status przerwania, to wywołanie metody `sleep` spowoduje wystąpienie wyjątku `Interrup-`

tedException. Dlatego też, gdy metoda sleep wywoływana jest w pętli, to nie musimy sprawdzać statusu przerwania wątku. Wystarczy jedynie zająć się obsługą wyjątku InterruptedException. Metoda run posiada wtedy następującą postać:

```
public void run()
{
   try
   {
      . . .
      while (są działania do wykonania)
      {
         wykonaj te działania
         Thread.sleep(delay);
      }

   }
   catch(InterruptedException exception)
   {
      // wątek przerwany w trakcie metody sleep lub wait
   }
   finally
   {
      wątek „sprząta po sobie", jeśli to konieczne
   }
   // zakończenie wykonywania metody run i tym samym wątku
}
```

Wygenerowanie wyjątku InterruptedException przez metodę sleep kasuje status przerwania wątku.

Istnieją dwie podobne metody, interrupted oraz isInterrupted. Metoda interrupted jest statyczną metodą, która sprawdza, czy *bieżący* wątek został przerwany. Dodatkowo wywołanie metody interrupted kasuje status przerwania wątku. Natomiast metoda isInterrupted wywoływana jest dla konkretnej instancji wątku i umożliwia sprawdzenie statusu przerwania dowolnego wątku. Jej wywołanie nie zmienia tego statusu.

Często można spotkać fragmenty kodu, w których obsługa wystąpienia wyjątku InterruptedException jest pominięta, co widać poniżej:

```
try { sleep(delay); }
catch (InterruptedException exception) {} // ŹLE!
```

Nie wolno pisać programów w ten sposób! Jeśli nie wiemy, jak obsłużyć wyjątek w klauzuli catch, to zawsze mamy jeszcze dwie inne możliwości:

- W klauzuli catch wywołać Thread.currentThread().interrupt() w celu nadania wątkowi statusu przerwania, który może być sprawdzony przez kod wywołujący.

```
void mySubTask()
{
   . . .
   try { sleep(delay); }
   catch (InterruptedException e) { Thread.currentThread().interrupt(); }
   . . .
}
```

- Lepsze rozwiązanie polega na zadeklarowaniu metody jako `throws InterruptedException` i pominięciu bloku `try`. Dzięki temu kod wywołujący metodę (lub w końcu sama metoda `run`) będzie mógł obsłużyć wyjątek.

```
void mySubTask() throws InterruptedException
{
   . . .
   sleep(delay);
   . . .
}
```

java.lang.Thread 1.0

- `void interrupt()` wysyła żądanie przerwania wątku. Status przerwania wątku przyjmuje wtedy wartość `true`. Jeśli wykonanie wątku jest zablokowane przez wywołanie metody `sleep` lub `wait`, to występuje wyjątek `InterruptedException`.
- `static boolean interrupted()` sprawdza, czy *bieżący* wątek (czyli ten, który ją wywołał) otrzymał żądanie przerwania. Zwróćmy uwagę, że jest to metoda statyczna. Jej wywołanie posiada efekt uboczny w postaci skasowania statusu przerwania bieżącego wątku.
- `boolean isInterrupted()` sprawdza, czy wątek otrzymał żądanie przerwania. Nie powoduje skasowania statusu przerwania wątku w przeciwieństwie do metody `interrupted`.
- `static Thread currentThread()` zwraca obiekt klasy `Thread` reprezentujący aktualnie wykonywany wątek.

Stany wątków

Wątki mogą znajdować się w jednym z czterech stanów:

- nowym,
- wykonywalnym,
- zablokowanym,
- martwym.

Każdy z wymienionych stanów omówiomy w kolejnych podrozdziałach.

Nowe wątki

Kiedy tworzymy wątek za pomocą operatora `new` — na przykład `new Ball()` — nie jest on jeszcze wykonywany. Znajduje się wtedy w stanie *nowy* i kod w nim zawarty nie jest wykonywany. Zanim zostanie uruchomiony, system musi wykonać jeszcze pewne operacje.

Wątki wykonywalne

Po wywołaniu metody `start` wątek jest *wykonywalny*. Wykonywalny wątek nie musi być od razu wykonywany. Przydział procesora zależy bowiem od systemu operacyjnego. Kiedy zostanie przydzielony, mówimy, że wątek jest *wykonywany*. (Dokumentacja języka Java nie przewiduje w tym przypadku oddzielnego stanu wątku, wątek wykonywany znajduje się nadal w stanie wykonywalnym).

Wątek nie powinien być wykonywany w sposób ciągły. Pożądane jest, aby każdy z wątków cyklicznie zawieszał swoje wykonanie, dając tym samym szansę innym wątkom. Szczegóły szeregowania wątków zależą także od systemu operacyjnego. Systemy stosujące szeregowanie z wydziedziczaniem udostępniają każdemu wątkowi wykonywalnemu przedział czasu, w którym może on wykonywać swoje zadania. Gdy przedział ten zakończy się, system operacyjny *wywłaszcza* wątek, dając okazję wykonania kolejnemu wątkowi (patrz rysunek 1.4). Wybierając ten wątek, system operacyjny bierze pod uwagę *priorytety* wątków (patrz strona 36.).

Wszystkie współczesne systemy operacyjne stacji roboczych i serwerów stosują szeregowanie z wywłaszczaniem, natomiast proste urządzenia, takie jak na przykład telefony komórkowe, mogą używać szeregowania kooperacyjnego. W takim przypadku wątek przestaje być wykonywany dopiero wtedy, gdy sam wywoła odpowiednią metodę w rodzaju `sleep` czy `yield`.

W systemach posiadających wiele procesorów każdy procesor może wykonywać inny wątek. Mamy wtedy sytuację, w której wątki rzeczywiście wykonywane są równolegle. Jeśli jednak wątków jest więcej niż procesorów, to szeregowanie wątków znowu musi odbywać się z podziałem czasu.

Należy pamiętać, że wątek będący w stanie wykonywalnym w danym momencie może być wykonywany lub nie (dlatego stan ten określamy jako wykonywalny, a nie wykonywany).

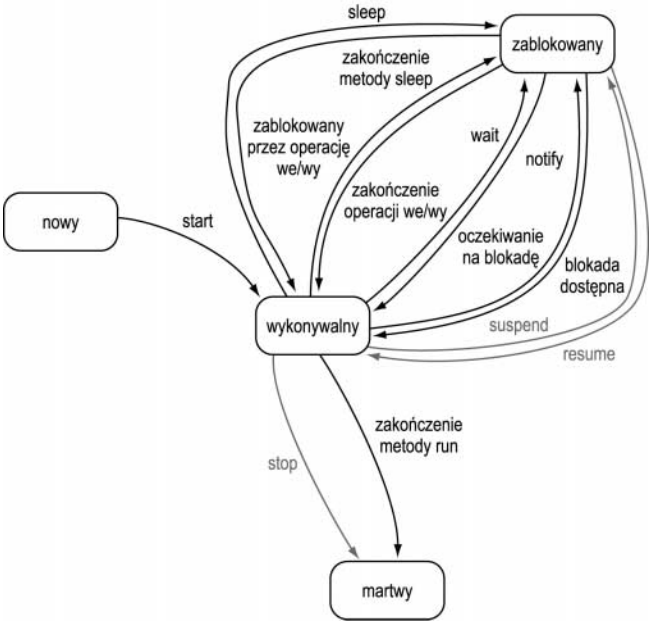
Wątki zablokowane

Wątek znajduje się w stanie *zablokowanym*, w przypadku gdy zaszła jedna z poniższych sytuacji:

- Wywołana została metoda `sleep` danego wątku.
- Wykonanie wątku zostało zablokowane przez operację wejścia-wyjścia, która zwróci sterowanie do wątku, gdy zakończy swoje działanie.
- Wątek próbuje zablokować dostęp do obiektu, który został już zablokowany przez inny wątek. Blokady obiektów omówimy w dalszej części tego rozdziału, na stronie 46.
- Wątek oczekuje na spełnienie pewnego warunku — patrz strona 49.
- Wywołana została metoda `suspend` danego wątku. Nie jest ona obecnie stosowana i nie powinna być używana w programach. Przyczynę tego stanu rzeczy omówimy w dalszej części tego rozdziału.

Rysunek 1.3 pokazuje diagram stanów wątku. Kiedy wątek zostaje zablokowany lub zakończony, rozpoczyna się wykonywanie kolejnego wątku. Kiedy zablokowany wątek zostaje ponownie aktywowany (ponieważ upłynął okres czasu, na który zawiesił on swoje wykonywanie bądź zakończyła się blokująca go operacja wejścia-wyjścia), program szeregujący wątki sprawdza, czy posiada on priorytet wyższy od aktualnie wykonywanego wątku. Jeśli tak, to *wywłaszcza* bieżący wątek i rozpoczyna wykonywanie reaktywowanego wątku.

Rysunek 1.3. *Diagram stanów wątku*



Wątek może powrócić ze stanu zablokowanego do stanu wykonywalnego, przebywając na diagramie jedną z następujących dróg.

1. Jeśli wątek zawiesił wykonywanie wywołując metodę `sleep`, musi upłynąć określona liczba milisekund.
2. Jeśli wątek został zablokowany przez operację wejścia-wyjścia, to musi ona się zakończyć.
3. Jeśli wątek zamierza uzyskać blokadę obiektu posiadaną przez inny wątek, to wątek ten musi zrzec się blokady obiektu. (Możliwe jest również oczekiwanie na zwolnienie blokady z określonym limitem czasu. Wtedy wątek zostaje odblokowany po jego upływie).
4. Jeśli wątek oczekuje na spełnienie pewnego warunku, to jego zajście musi zostać zasygnalizowane przez inny wątek. (Możliwe jest również oczekiwanie na spełnienie warunku z określonym limitem czasu. Wtedy wątek zostaje odblokowany po jego upływie).
5. Jeśli wątek zawiesił swoje wykonywanie, to inny wątek musi wywołać jego metodę `resume`. Stosowanie metod `suspend` i `resume` nie jest jednak zalecane.

Zablokowany wątek może powrócić do stanu wykonywalnego jedynie tą samą drogą, na której został zablokowany. W szczególności nie wystarczy wywołanie metody `resume`, aby odblokować wątek.

> Dla odblokowania operacji wejścia i wyjścia należy stosować mechanizm *kanałów* oferowany przez nową bibliotekę wejścia i wyjścia. Gdy inny wątek zamyka kanał, to zablokowany wątek powraca do stanu wykonywalnego, a blokująca operacja generuje wyjątek `ClosedChannelException`.

Wątki martwe

Wątek może być martwy gdy zaistnieje jeden z dwóch powodów.

- Zakończy się wykonywanie metody `run` w prawidłowy sposób.
- Wykonanie metody `run` zostanie przerwane ze względu na wystąpienie wątku, którego metoda ta nie obsługuje.

W szczególności możliwe jest „zabicie" wątku przez wywołanie metody `stop`. Powoduje ona wystąpienie błędu `ThreadDeath`, co kończy wykonywanie wątku. Metoda ta niesie ze sobą pewne zagrożenia (omówimy je w dalszej części rozdziału) i nie powinna być stosowana w programach.

Aby uzyskać informację o tym, czy dany wątek „żyje" (czyli jest wykonywalny lub zablokowany), stosujemy metodę `isAlive`. Zwraca ona wartość logiczną `true`, gdy wątek jest wykonywalny lub zablokowany albo wartość `false`, jeśli jest nowy bądź martwy.

> Nie można uzyskać informacji o tym, czy dany wątek jest wykonywalny, czy też zablokowany, a także informacji o tym, czy jest w danej chwili wykonywany. Podobnie nie można rozróżnić stanu wątku, w którym nie jest on jeszcze wykonywalny, od stanu, w którym jest już martwy.

`java.lang.Thread 1.0`

- `boolean isAlive()` zwraca wartość `true`, jeśli wątek już rozpoczął działanie i jeszcze się nie zakończył.
- `void stop()` kończy wykonanie wątku. Użycie tej metody nie jest zalecane.
- `void suspend()` zawiesza wykonanie wątku. Użycie tej metody nie jest zalecane.
- `void resume()` podejmuje wykonanie wątku zawieszonego metodą `suspend`. Użycie obu metod nie jest zalecane.
- `void join()` oczekuje zakończenia działania określonego wątku.
- `void join(long millis)` oczekuje zakończenia działania określonego wątku lub upływu `millis` milisekund.

Właściwości wątków

W kolejnych punktach omówione zostaną inne właściwości wątków: priorytety, wątki-demony, grupy wątków i domyślne procedury obsługi wyjątków.

Priorytety wątków

W języku Java z każdym wątkiem związany jest określony *priorytet*. Domyślnie wątek dziedziczy priorytet po swoim wątku nadrzędnym, czyli wątku, który go utworzył. Można go jednak zmniejszyć lub zwiększyć, stosując metodę `setPriority`. Za jej pomocą można priorytetowi nadać dowolną wartość z przedziału od `MIN_PRIORITY` (zdefiniowane jako 1 w klasie `Thread`) do `MAX_PRIORITY` (zdefiniowane jako 10). Priorytet `NORM_PRIORITY` posiada wartość 5.

Gdy program szeregowania wątków musi wybrać nowy wątek, zawsze wybiera wątek wykonywalny o najwyższym priorytecie. Należy zaznaczyć, że wszelkie zasady działania związane z priorytetami wątków są w znacznym stopniu zależne od konkretnego systemu operacyjnego. Jeśli maszyna wirtualna języka Java wykorzystuje zarządzanie wątkami danej platformy, to musi dokonać odwzorowania priorytetów wątków zdefiniowanych w języku Java na poziomy priorytetów tej platformy (których może być mniej lub więcej niż wartości priorytetów w języku Java).

Na przykład w systemie Windows NT/XP istnieje siedem poziomów priorytetów, mniej niż definiuje to Java. Niezależnie więc od sposobu odwzorowanie priorytetów zdarzać się będzie, że wątki, które mają różny priorytet w języku Java, będą posiadać w rzeczywistości ten sam priorytet z punktu widzenia systemu operacyjnego. W przypadku maszyny wirtualnej Java firmy Sun dla systemu Linux informacja o priorytetach wątków jest w ogóle ignorowana.

Z tego powodu należy traktować priorytety wątków jedynie jako rodzaj wskazówki dla programu szeregującego. W żadnym wypadku nie należy tworzyć programów, których poprawne działanie zależy od poziomów priorytetów.

> Jeśli zamierzasz używać priorytetów wątków, to powinieneś wiedzieć o błędzie popełnianym często przez początkujących. Jeśli w programie istnieje kilka wątków o wysokim priorytecie, które rzadko znajdują się w stanie zablokowanym, to mogą one uniemożliwić wykonywanie wątków o niskim priorytecie. Program szeregujący wątki zawsze wybiera najpierw spośród wątków o wyższym priorytecie, nawet jeśli w ten sposób wątki o niższym priorytecie nie są nigdy wykonywane.

`java.lang.Thread 1.0`

- `void setPriority(int newPriority)` określa priorytet wątku. Wartość priorytetu musi znajdować się w przedziale od `Thread.MIN_PRIORITY` do `Thread.MAX_PRIORITY`. Priorytet zwykłego wątku posiada wartość `Thread.NORM_PRIORITY`.
- `static int MIN_PRIORITY` minimalna wartość priorytetu wątku (wynosi 1).
- `static int NORM_PRIORITY` domyślna wartość priorytetu wątku (wynosi 5).

- `static int MAX_PRIORITY` maksymalna wartość priorytetu wątku (wynosi 10).
- `static void yield()` zawiesza wykonanie bieżącego wątku. Kolejnym wykonywanym wątkiem będzie wątek wykonywalny, o co najmniej takim samym priorytecie jak wątek zawieszony. Metoda statyczna.

Wątki-demony

Wątek staje się *wątkiem-demonem* przez wywołanie

```
t.setDaemon(true);
```

Wątek-demon nie ma w sobie nic demonicznego. Jest to po prostu wątek, którego jedynym zadaniem jest służenie innym wątkom. Przykładem wątków-demonów mogą być wątki zegara, które dostarczają innym wątkom informacji o upływie czasu. Jeśli dla danego programu pozostaje jedynie wątek-demon, to program kończy swoje działania.

`java.lang.Thread 1.0`

- `void setDaemon(boolean isDaemon)` oznacza dany wątek jako wątek-demon bądź wątek użytkownika. Jeśli aktywne są tylko wątki-demony, to program kończy swoje działanie. Metoda ta musi być wywołana przed wystartowaniem wątku.

Grupy wątków

Niektóre programy wykonują wiele wątków. Przydatna wtedy okazuje się możliwość kategoryzacji na podstawie ich funkcji. Jako przykład weźmy przeglądarkę internetową. Jeśli wiele wątków zajmuje się ładowaniem różnych obrazków zamieszczonych na danej stronie, a użytkownik wybierze przycisk *Stop*, to wykonywanie wszystkich tych wątków powinno zostać przerwane. Java pozwala tworzyć w tym celu *grupy wątków*.

Grupę wątków tworzymy, wywołując konstruktor klasy `ThreadGroup`:

```
String groupName = . . .;
ThreadGroup g = new ThreadGroup(groupName);
```

Łańcuch znaków będący parametrem konstruktora identyfikuje grupę wątków i musi być unikalny. Wątki dodajemy do danej grupy, określając ją za pomocą parametru konstruktora wątku.

```
Thread t = new Thread(g, threadName);
```

Aby dowiedzieć się, czy istnieje jakiś wykonywalny wątek danej grupy, używamy metody `activeCount`.

```
if (g.activeCount() == 0)
{
   // wszystkie wątki danej grupy zakończyły działanie
}
```

Aby przerwać wykonywanie wszystkich wątków danej grupy, wywołujemy metodę `interrupt` obiektu reprezentującego daną grupę.

```
g.interrupt(); // przerywa wszystkie wątki w grupie g
```

Ten sam efekt możemy osiągnąć bez używania grup wątków, lecz za pomocą egzekutorów (patrz strona 88.).

Grupy wątków mogą posiadać podgrupy. Domyślnie nowo tworzona grupa wątków staje się podgrupą bieżącej grupy. Grupę nadrzędną można także określić za pomocą parametru konstruktora grupy (patrz opis konstruktora). Metody `activeCount` oraz `interrupt` odnoszą się do wszystkich wątków w grupie oraz jej podgrup.

java.lang.Thread 1.0

- `Thread(ThreadGroup g, String name)` tworzy nowy wątek należący do danej grupy wątków.

  Parametry: `g` grupa wątków, do której należeć będzie nowy wątek,

  `name` nazwa nowego wątku.

- `ThreadGroup getThreadGroup()` zwraca grupę wątków, do której należy dany wątek.

java.lang.ThreadGroup 1.0

- `ThreadGroup(String name)` tworzy nową grupę wątków. Grupa ta stanowić będzie podgrupę grupy, do której należy bieżący wątek.

  Parametry: `name` nazwa nowej grupy wątków.

- `ThreadGroup(ThreadGroup parent, String name)` tworzy nową grupę wątków.

  Parametry: `parent` grupa nadrzędna tworzonej grupy,

  `name` nazwa nowej grupy wątków.

- `int activeCount()` zwraca liczbę wątków aktywnych danej grupy.
- `int enumerate(Thread[] list)` pobiera referencje do wszystkich wątków aktywnych w danej grupie; aby określić rozmiar tablicy referencji, warto użyć metody `activeCount`; może zdarzyć się, że tablica okaże się za mała (na przykład gdy po użyciu metody `activeCount` utworzono dużo nowych wątków), wtedy znajdą się w niej referencje do części wątków.

  Parametry: `list` tablica, w której mają znaleźć się referencję wątków.

- `ThreadGroup getParent()` pobiera grupę nadrzędną danej grupy.
- `void interrupt()` przerywa wszystkie wątki danej grupy i jej wszystkich podgrup.

Procedury obsługi wyjątków

Metoda `run` wątku nie może wygenerować weryfikowalnego wyjątku, ale jej działanie może zostać zakończone przez wyjątek nieweryfikowalny. W takim przypadku wątek kończy swoje istnienie.

W sytuacji tej nie istnieje klauzula `catch`, do której mogłaby zostać przekazana obsługa wyjątku. Dlatego też, zanim wątek zakończy działanie, obsługa wyjątku zostaje przekazana do procedury obsługi wyjątków.

Procedura ta musi należeć do klasy, która implementuje interfejs `ThreadUncaughtExceptionHandler`. Interfejs ten składa się z pojedynczej metody

```
void uncaughtException(Thread t, Throwable e)
```

Począwszy od JDK 5.0 metoda `setUncaughtExceptionHandler` pozwala nam zainstalować własną procedurę obsługi wyjątków dla dowolnego wątku. Metoda statyczna `setDefaultUncaughtExceptionHandler` należąca do klasy `Thread` umożliwia zainstalowanie domyślnej procedury obsługi wyjątków dla wszystkich wątków programu. Procedura taka może, na przykład, używać interfejsu programowego dzienników do tworzenia raportów o wyjątkach nieobsłużonych.

Jeśli nie zainstalujemy domyślnej procedury obsługi wyjątków dla wszystkich wątków, to będzie ona `null`. Natomiast gdy nie zainstalujemy takiej procedury dla poszczególnych wątków, to procedura ta należeć będzie do obiektu `ThreadGroup` dla danego wątku.

Klasa `ThreadGroup` implementuje interfejs `Thread.UncaughtExceptionHandler`. Metoda `uncaughtException` tej klasy wykonuje podejmuje następujące działania:

1. Jeśli dana grupa wątków należy do grupy nadrzędnej, to wywoływana jest metoda `uncaughtException` grupy nadrzędnej.
2. W przeciwnym razie, jeśli metoda `Thread.getDefaultExceptionHandler` zwraca procedurę obsługi różną od `null`, to zostaje ona wywołana.
3. W przeciwnym razie, gdy `Throwable` jest instancją `ThreadDeath`, nie jest wykonywana żadna operacja.
4. W przeciwnym razie nazwa wątku i zrzut stosu dla `Throwable` zostają zapisane w `System.err`.

Wspomniany zrzut stosu znany jest dobrze wszystkim piszącym programy w języku Java.

> W wersjach wcześniejszych niż JDK 5.0 nie istniała bezpośrednia możliwość zainstalowania własnej procedury obsługi wyjątków dla wszystkich wątków ani dla poszczególnych wątków. W tym celu należało stworzyć klasę pochodną klasy `ThreadGroup` i przesłonić jej metodę `uncaughtException` własną wersją.

java.lang.Thread 1.0

- `static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` 5.0
- `static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()` 5.0 konfiguruje lub pobiera domyślną procedurę obsługi wyjątków.
- `void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` 5.0
- `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()` 5.0 konfiguruje lub pobiera procedurę obsługi wyjątków. Jeśli żadna taka procedura nie zostanie skonfigurowana, to używana jest w tym celu grupa, do której należy wątek.

java.lang.Thread.UncaughtExceptionHandler 5.0

- `void uncaughtException(Thread t, Throwable e)`. Metodę tę definiujemy, aby uzyskać raport, gdy działanie wątku zostało zakończone na skutek nieobłużonego wyjątku.

  Parametry: `t` wątek zakończony na skutek wyjątku, który nie został obsłużony,

  `e` wyjątek, który nie został obsłużony.

java.lang.ThreadGroup 1.0

- `void uncaughtException(Thread t, Throwable e)`. Metodę tę zastępujemy własną wersją, aby reagować na wyjątki powodujące zakończenie wątków tej grupy. Domyślna implementacja wywołuje tę samą metodę grupy nadrzędnej, a jeśli grupa nadrzędna nie istnieje, to wysyła zrzut stosu do standardowego strumienia błędów. (Jeśli `e` jest obiektem klasy `ThreadDeath`, to informacja o zawartości stosu nie jest wysyłana. Obiekty klasy `ThreadDeath` generowane są przez metodę `stop`, której nie należy stosować).