PRENTICE HALL

JAVA®

Podstawy

WYDANIE XI

CAY S. HORSTMANN

Helion

Tytuł oryginalny: Core Java Volume I - Fundamentals (11th Edition)

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-5778-5

Authorized translation from the English language edition, entitled CORE JAVA VOLUME I – FUNDAMENTALS, 1, 11th Edition by HORSTMANN, CAY S., published by Pearson Education, Inc, publishing as Prentice Hall.

Helion SA
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)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/javp11
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/javp11.zip

Printed in Poland.

Spis treści

12

Współbieżność

W tym rozdziale:

Większość użytkowników systemów operacyjnych zna pojęcie **wielozadaniowości**, czyli zdolności systemu do uruchamiania więcej niż jednego programu pozornie jednocześnie. Można na przykład pisać lub wysyłać e-mail i w tym samym czasie drukować jakiś dokument. W dzisiejszych czasach coraz częściej spotyka się komputery wyposażone w więcej niż jeden procesor, choć liczba procesów działających jednocześnie nie jest ograniczona liczbą procesorów. System operacyjny stwarza pozory równoległego wykonywania zadań, każdemu procesowi przydzielając odpowiedni czas pracy procesora.

Programy wielowątkowe przenoszą koncepcję wielozadaniowości o jeden poziom niżej, gdzie poszczególne programy sprawiają złudzenie wykonywania wielu zadań naraz. Każde z tych zadań jest zwyczajowo nazywane **wątkiem** (ang. *thread*), a pełna nazwa to wątek sterowania (ang. *thread of control*). Programy potrafiące działać w więcej niż jednym wątku nazywają się programami **wielowątkowymi** (ang. *multithreaded*).

Jaka jest zatem różnica pomiędzy wieloma *procesami* a wieloma *wątkami*? Przede wszystkim należy zauważyć, że każdy proces posiada pełen zestaw własnych zmiennych, podczas gdy wątki współdzielą dane z innymi wątkami. Brzmi to dosyć ryzykownie i rzeczywiście może czasami sprawiać problemy, o czym przekonasz się za chwilę. Z drugiej strony dzięki współdzieleniu zmiennych komunikacja pomiędzy wątkami zachodzi sprawniej i jest łatwiejsza do

zaprogramowania niż komunikacja międzyprocesowa. Ponadto wątki w niektórych systemach operacyjnych są lżejsze od procesów, to znaczy utworzenie i zniszczenie pojedynczego wątku zajmuje mniej czasu niż uruchomienie nowego procesu.

Wielowątkowość jest niezwykle praktycznym narzędziem. Wiadomo na przykład, że przeglądarka powinna mieć możliwość pobierania kilku obrazów jednocześnie, a serwer sieciowy musi obsługiwać wiele żądań w tym samym czasie. Programy z graficznym interfejsem użytkownika dysponują osobnym wątkiem do zbierania zdarzeń z interfejsu pochodzących od środowiska operacyjnego. Ten rozdział dotyczy pisania aplikacji wielowątkowych w Javie.

Ostrzeżenie: programy wielowątkowe bywają bardzo skomplikowane. My opisujemy wszystkie narzędzia, których może potrzebować programista. Jednak w poszukiwaniu opisów bardziej zaawansowanych technik programowania systemowego odsyłamy do innych źródeł, na przykład książki *Java. Współbieżność dla praktyków*, której autorami są Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes i Doug Lea (Helion, Gliwice 2007).

12.1. Czym są wątki

Na początek przyjrzymy się prostemu programowi, który wykorzystuje dwa wątki do przenoszenia pieniędzy między kontami bankowymi. Będziemy używać klasy `Bank` zapisującej salda określonej liczby kont. Metoda `transfer` przelewa określoną kwotę z jednego konta na inne. Implementacja jest pokazana na listingu 12.2.

W pierwszym wątku przelejemy pieniądze z konta `0` na konto `1`. Drugi wątek będzie przelewał pieniądze z konta `2` na konto `3`.

Poniżej znajduje się prosta procedura uruchamiania zadania w osobnym wątku:

1. Umieść kod zadania w metodzie `run` klasy implementującej interfejs `Runnable`. Ten bardzo prosty interfejs zawiera tylko jedną metodę:

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

   Dzięki temu, że `Runnable` to interfejs funkcyjny, można utworzyć egzemplarz za pomocą wyrażenia lambda:

   ```
   Runnable r = () -> { task code };
   ```

2. Utwórz obiekt `Thread` z egzemplarza `Runnable`:

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

3. Uruchom wątek:

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

   Aby utworzyć osobny wątek do przelewania pieniędzy, należy umieścić kod realizujący przelew w metodzie `run` egzemplarza `Runnable`, a następnie uruchomić wątek:

```
Runnable r = () -> {
  try
  {
    for (int i = 0; i < STEPS; i++)
    {
      double amount = MAX_AMOUNT * Math.random();
      bank.transfer(0, 1, amount);
      Thread.sleep((int) (DELAY * Math.random()));
    }
  }
  catch (InterruptedException e)
  {
  }
};
var t = new Thread(r);
t.start();
```

W określonej liczbie kroków wątek ten przelewa losową kwotę, po czym zasypia na losową ilość czasu.

Musimy przechwycić wyjątek `InterruptedException`, którego zgłoszeniem grozi metoda `sleep`.

Dokładniej tym wyjątkiem zajmujemy się w punkcie 12.3.1, „Przerywanie wątków”. Pracę wątków zazwyczaj przerywa się w celu zakończenia ich działania. W chwili wystąpienia wyjątku `InterruptedException` nasza metoda `run` także zakończy działanie.

Ponadto nasz program uruchamia drugi wątek, który przelewa pieniądze z konta 2 na konto 3. Poniżej znajduje się wynik działania tego programu:

```
Thread[Thread-1,5,main] 606,77 z 2 na 3 Saldo całkowite: 400000,00
Thread[Thread-0,5,main]  98,99 z 0 na 1 Saldo całkowite: 400000,00
Thread[Thread-1,5,main] 476,78 z 2 na 3 Saldo całkowite: 400000,00
Thread[Thread-0,5,main] 653,64 z 0 na 1 Saldo całkowite: 400000,00
Thread[Thread-1,5,main] 807,14 z 2 na 3 Saldo całkowite: 400000,00
Thread[Thread-0,5,main] 481,49 z 0 na 1 Saldo całkowite: 400000,00
Thread[Thread-0,5,main] 203,73 z 0 na 1 Saldo całkowite: 400000,00
Thread[Thread-1,5,main] 111,76 z 2 na 3 Saldo całkowite: 400000,00
Thread[Thread-1,5,main] 794,88 z 2 na 3 Saldo całkowite: 400000,00
. . .
```

Wyniki dwóch wątków przeplatają się, co wskazuje na to, że wątki te są wykonywane współbieżnie. Czasami jednak wyniki takiej przeplatanki mogą być mniej uporządkowane.

To wszystko! Od tej pory umiesz wykonywać zadania współbieżnie. W pozostałej części tego rozdziału znajduje się opis technik kontroli interakcji między wątkami.

Na listingu 12.1 znajduje się kompletny kod.

Listing 12.1. threads/ThreadTest.java

```
package threads;

/**
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
```

```
 */
public class ThreadTest
{
   public static final int DELAY = 10;
   public static final int STEPS = 100;
   public static final double MAX_AMOUNT = 1000;

   public static void main(String[] args)
   {
      var bank = new Bank(4, 100000);
      Runnable task1 = () ->
      {
         try
         {
            for (int i = 0; i < STEPS; i++)
            {
               double amount = MAX_AMOUNT * Math.random();
               bank.transfer(0, 1, amount);
               Thread.sleep((int) (DELAY * Math.random()));
            }
         }
         catch (InterruptedException e)
         {
         }
      };

      Runnable task2 = () ->
      {
         try
         {
            for (int i = 0; i < STEPS; i++)
            {
               double amount = MAX_AMOUNT * Math.random();
               bank.transfer(2, 3, amount);
               Thread.sleep((int) (DELAY * Math.random()));
            }
         }
         catch (InterruptedException e)
         {
         }
      };

      new Thread(task1).start();
      new Thread(task2).start();
   }
}
```

Wątek można też zdefiniować poprzez utworzenie podklasy klasy `Thread`, np.:

```
class MyThread extends Thread
 {
   public void run()
   {
     task code
   }
 }
```

Następnie można utworzyć obiekt podklasy i wywołać jego metodę `start`. Technika ta nie jest jednak już zalecana, ponieważ powinno się oddzielać zadanie do wykonania równoległego od mechanizmu wykonywania. Jeśli zadań jest dużo, tworzenie osobnego wątku dla każdego z nich jest zbyt kosztowne. W takim przypadku lepiej użyć puli wątków — patrz punkt 12.6.2, „Klasa Executors”.

Nie wywołuj metody `run` klasy `Thread` ani obiektu typu `Runnable`. Jej bezpośrednie wywołanie powoduje tylko wykonanie zadania i tym samym wątku — nie zostaje uruchomiony nowy wątek. Zamiast tego należy wywołać metodę `Thread.start`, która tworzy nowy wątek wykonujący metodę `run`.

Listing 12.2. threads/Bank.java

```
package threads;

import java.util.*;

/**
 * Bank z pewną liczbą kont.
 */
public class Bank
{
   private final double[] accounts;

   /**
    * Tworzy bank.
    * @param n liczba kont
    * @param initialBalance początkowe saldo każdego konta
    */
   public Bank(int n, double initialBalance)
   {
      accounts = new double[n];
      Arrays.fill(accounts, initialBalance);
   }

   /**
    * Przelewa pieniądze z jednego konta na inne.
    * @param from konto źródłowe
    * @param to konto docelowe
    * @param amount kwota przelewu
    */
   public void transfer(int from, int to, double amount)
   {
      if (accounts[from] < amount) return;
      System.out.print(Thread.currentThread());
      accounts[from] -= amount;
      System.out.printf(" %10.2f z %d na %d", amount, from, to);
      accounts[to] += amount;
      System.out.printf(" Saldo całkowite: %10.2f%n", getTotalBalance());
   }

   /**
    * Oblicza sumę wszystkich sald kont.
    * @return saldo całkowite
    */
```

```
    public double getTotalBalance()
    {
       double sum = 0;

       for (double a : accounts)
          sum += a;

       return sum;
    }

    /**
     * Sprawdza liczbę kont w banku.
     * @return liczba kont
     */
    public int size()
    {
       return accounts.length;
    }
}
```

`java.lang.Thread` **1.0**

- `Thread(Runnable target)`

  Tworzy nowy wątek, który wywołuje metodę `run()` wskazanego obiektu.

- `void start()`

  Uruchamia ten wątek, powodując wywołanie metody `run()`. Metoda ta zwraca wartość natychmiast. Nowy wątek działa współbieżnie.

- `void run()`

  Wywołuje metodę `run` powiązanego obiektu typu `Runnable`.

- `static void sleep(long millis)`

  Zatrzymuje wykonywanie na określoną liczbę milisekund.

`java.lang.Runnable` **1.0**

- `void run()`

  Metoda, która musi zostać przesłonięta oraz otrzymać wytyczne dla zadania, które ma zostać wykonane.

12.2. Stany wątków

Wątek może być w jednym z sześciu stanów:

- `NEW` (nowy),
- `RUNNABLE` (wykonywalny),
- `BLOCKED` (zablokowany),

- WAITING (oczekujący),
- TIMED WAITING (oczekujący określoną ilość czasu),
- TERMINATED (zakończony).

Znaczenie tych wszystkich stanów zostało opisane poniżej.

Do sprawdzania aktualnego stanu wątku służy metoda getState.

12.2.1. Wątki tworzone za pomocą operatora new

Wątek utworzony za pomocą operatora new — na przykład new Thread(r) — nie jest od razu uruchamiany. Oznacza to, że pozostaje on w stanie NEW. Jeśli wątek znajduje się w stanie NEW, program nie zaczął jeszcze wykonywać znajdującego się w nim kodu. Przed uruchomieniem wątku trzeba wykonać jeszcze kilka dodatkowych czynności.

12.2.2. Wątki RUNNABLE

Po wywołaniu metody start wątek przechodzi w stan RUNNABLE. Wątek taki może, ale nie musi być uruchomiony. Przydział czasu dla wątku leży w gestii systemu operacyjnego (w Javie dla stanu działania wątku nie wprowadzono osobnej nazwy, dlatego uruchomiony wątek nadal pozostaje w stanie RUNNABLE).

Po uruchomieniu wątek nie musi działać cały czas. Zaleca się nawet wstrzymywanie działających wątków co jakiś czas, aby dać szansę na działanie innym wątkom. Szczegółowa kontrola harmonogramu wykonywania wątków zależy od usług udostępnianych przez system operacyjny. Systemy planowania wywłaszczającego wątków przydzielają każdemu wykonywalnemu wątkowi określoną ilość czasu na wykonanie zadania. Kiedy czas mija, system operacyjny *wywłaszcza* wątek i przydziela czas innemu wątkowi (zobacz rysunek 12.2). Przy wybieraniu kolejnego wątku system operacyjny kieruje się *priorytetami* wątków — zobacz punkt 12.3.5, „Priorytety wątków”.

Wszystkie nowoczesne systemy operacyjne — zarówno serwerowe, jak i przeznaczone na komputery osobiste — stosują planowanie wywłaszczające. Mniejsze urządzenia, jak telefony komórkowe, mogą wykorzystywać planowanie kooperacyjne. W takim urządzeniu wątek traci sterowanie, jeśli wywoła metodę yield lub zostanie zablokowany czy przestawiony w stan oczekiwania.

W komputerach z kilkoma procesorami każdy procesor może wykonywać osobny wątek, dzięki czemu wiele wątków może działać równolegle. Oczywiście jeśli wątków jest więcej niż procesorów, system planujący i tak musi się zająć przydzielaniem czasu.

Należy zawsze pamiętać, że wykonywalny wątek może w danej chwili być uruchomiony lub nie (dlatego stan ten nazwano RUNNABLE, co oznacza „wykonywalny”, zamiast na przykład RUNNING, co znaczyłoby „wykonywany”).

`java.lang.Thread` **1.0**

- `static void yield()`

  Zmusza aktualnie wykonywany wątek do oddania kontroli innemu wątkowi. Uwaga: to jest metoda statyczna.

12.2.3. Wątki BLOCKED i WAITING

Kiedy wątek znajduje się w stanie `BLOCKED` (zablokowany) lub `WAITING` (oczekujący), jest okresowo nieaktywny. Nie wykonuje żadnego kodu i zużywa minimalne ilości zasobów. Decyzja o jego reaktywacji należy do algorytmu planującego, który uzależnia ją od sposobu, w jaki wątek wszedł w stan nieaktywności:

- Kiedy wątek usiłuje założyć blokadę wewnętrzną na obiekt (ale nie utworzyć obiekt klasy `Lock` z biblioteki `java.util.concurrent`), który jest aktualnie w dyspozycji innego wątku, zostaje *zablokowany* (blokady `java.util.concurrent` opisujemy w punkcie 12.4.3, „Obiekty klasy Lock”, a blokady wewnętrzne obiektów w punkcie 12.4.5, „Słowo kluczowe synchronized”). Wątek zostaje odblokowany, gdy wszystkie pozostałe wątki zwolnią blokadę, a algorytm planujący zezwoli mu na przejęcie tego obiektu.
- Kiedy wątek czeka, aż inny wątek powiadomi algorytm planujący o warunku, wchodzi w stan `WAITING` (oczekujący). Warunki opisujemy w punkcie 12.4.4, „Warunki”. Stan ten jest wyzwalany przez wywołanie metody `Object.wait` lub `Thread.join` bądź oczekiwanie na obiekt klasy `Lock` lub `Condition` z biblioteki `java.util.concurrent`. W praktyce różnica pomiędzy stanem zablokowania a oczekiwania nie jest jasna.
- Niektóre metody posiadają parametr określający długość czasu wykonywania. Ich wywołanie powoduje wejście wątku w stan `TIMED WAITING`. Stan ten utrzymuje się, aż upłynie określony czas lub wątek odbierze odpowiednie powiadomienie. Do tego typu metod zalicza się metodę `Thread.sleep` oraz czasowe wersje metod `Object.wait`, `Thread.join`, `Lock.tryLock` i `Condition.wait`.

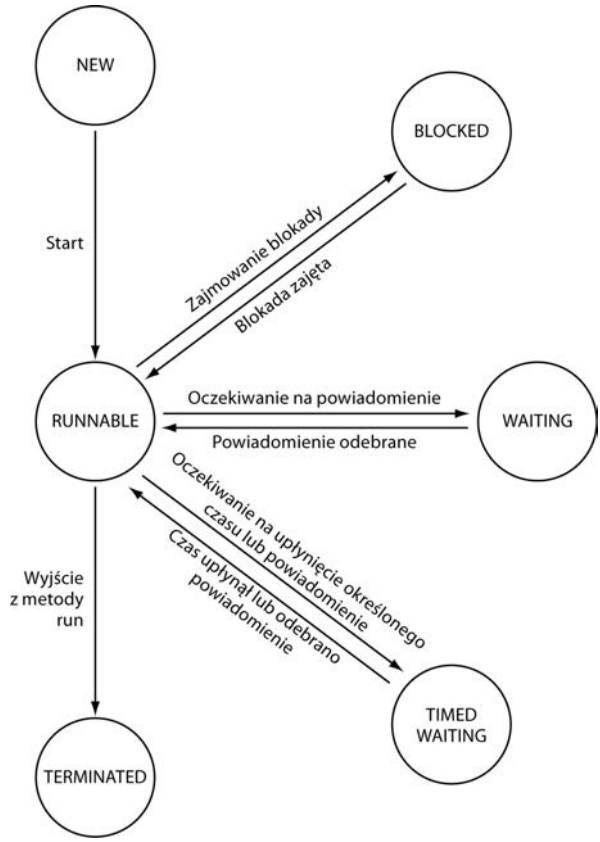
Rysunek 12.1 przedstawia stany, w których może się znaleźć wątek, oraz możliwe przejścia pomiędzy poszczególnymi stanami. Kiedy wątek zostanie zablokowany lub przejdzie w stan oczekiwania (bądź zostanie zakończony), planowane jest uruchomienie kolejnego wątku. Kiedy wątek jest reaktywowany (ponieważ skończył się jego czas oczekiwania lub zajął blokadę), algorytm planujący sprawdza, czy ma on wyższy priorytet niż aktualnie działające wątki. Jeśli tak, wywłaszcza jeden z uruchomionych wątków i uruchamia nowy wątek.

12.2.4. Zamykanie wątków

Wątek może zostać zamknięty na jeden z dwóch sposobów:

- naturalnie wraz z zakończeniem metody `run`;
- niespodziewanie z powodu nieprzechwyconego wyjątku, który zakończył metodę `run`.

Rysunek 12.1.
Stany wątków



Do zamykania wątków służy metoda `stop`. Wyrzuca ona błąd `ThreadDeath`, który zabija wątek. Metoda ta jest jednak odradzana, dlatego nie powinno się jej już używać.

`java.lang.Thread` **1.0**

- `void join()`

  Oczekuje na zamknięcie określonego wątku.

- `void join(long millis)`

  Czeka na zamknięcie określonego wątku albo na upływ określonej liczby milisekund.

- `Thread.State getState()` **5.0**

  Zwraca stan wątku: `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING` lub `TERMINATED`.

- `void stop()`

  Zatrzymuje wątek. Metoda ta jest odradzana.

- `void suspend()`

  Zawiesza wykonywanie wątku. Metoda odradzana.

- `void resume()`

  Ponownie uaktywnia wątek. Metoda ta może działać tylko po wywołaniu metody `suspend()`. Metoda odradzana.

12.3. Własności wątków

W tym podrozdziale opisujemy różne własności wątków: stan przerwania, wątki demony, procedury obsługi nieprzechwyconych wyjątków oraz garść starych elementów, których nie powinno się już używać.

12.3.1. Przerywanie wątków

Wątek kończy działanie w chwili zwrócenia przez jego metodę `run` wartości w wyniku wywołania instrukcji `return`, po wykonaniu ostatniej instrukcji w ciele metody `run` lub jeśli wystąpi nieprzechwycony w tej metodzie wyjątek. W pierwszej wersji Javy istniała jeszcze metoda `stop`, którą mógł wywołać jeden wątek w celu zamknięcia innego wątku. Jest ona jednak obecnie odradzana. Powody tego stanu rzeczy opisujemy w punkcie 12.4.13, „Dlaczego metody stop i suspend są wycofywane".

Jedynym sposobem na *zmuszenie* wątku do zakończenia działania jest wywołanie wycofywanej metody `stop`. Natomiast za pomocą metody `interrupt` można *poprosić* o zamknięcie wątku.

Wywołanie metody `interrupt` na rzecz wątku powoduje ustawienie jego **statusu przerwania** (ang. *interrupted state*). Jest to zmienna logiczna obecna w każdym wątku. Każdy wątek powinien co jakiś czas sprawdzać, czy nie został przerwany.

Aby sprawdzić, czy status przerwania został ustawiony, należy najpierw pobrać bieżący wątek za pomocą metody `Thread.currentThread`, a następnie wywołać metodę `isInterrupted`:

```
while (!Thread.currentThread().isInterrupted() && więcej instrukcji)
{
  dodatkowe działania
}
```

Statusu przerwania nie można jednak sprawdzić, jeśli wątek jest zablokowany. W takiej sytuacji do gry wchodzi wyjątek `InterruptedException`. Kiedy metoda `interrupt` jest wywoływana na rzecz wątku, który blokują metody takie jak `sleep` czy `wait`, wywołania blokujące zostają zakończone przez wyjątek `InterruptedException` (istnieją metody blokujące wejścia-wyjścia, których nie można przerwać; w takiej sytuacji należy rozważyć użycie ich przerywalnych zamienników — szczegółowe informacje na ten temat znajdują się w rozdziałach 2. i 4. drugiego tomu).

Nie istnieje żaden wymóg formalny, że wątek, który został przerwany, musi zostać zamknięty. Przerwanie wątku powoduje jedynie zwrócenie jego uwagi, a decyzja, jak na to zareagować, należy do niego samego. Niektóre bardzo ważne wątki powinny obsłużyć wyjątek i kontynuować pracę. Często jednak przerwanie jest przez wątek interpretowane jako żądanie zamknięcia. Metoda `run` takiego wątku wygląda następująco:

```
Runnable r = () -> {
   try
   {
      . . .
      while (!Thread.currentThread().isInterrupted() && więcej zadań)
      {
         jakieś czynności
      }
   }
   catch(InterruptedException e)
   {
      // działanie wątku przerwano w czasie uśpienia lub oczekiwania
   }
   finally
   {
      sprzątanie w razie potrzeby
   }
   // koniec metody run oznacza koniec wątku
};
```

Wywoływanie metody `isInterrupted` staje się bezcelowe, jeśli po każdej iteracji wywoływana jest metoda `sleep` (lub inna pozwalająca na przerwanie). Jeśli metoda `sleep` zostanie wywołana na rzecz wątku z ustawionym statusem przerwania, wątek ten nie zostanie uśpiony. Zamiast tego jego status zostanie wyzerowany oraz zostanie zgłoszony wyjątek `Interrupted` ➥`Exception`. Dlatego jeśli pętla zawiera wywołanie metody `sleep`, nie należy w niej sprawdzać statusu przerwania. W zamian należy przechwycić wyjątek `InterruptedException`:

```
Runnable r = () -> {
  try
  {
    . . .
    while (instrukcje)
    {
      instrukcje
      Thread.sleep(delay);
    }
  }
  catch(InterruptedException e)
  {
    // Wątek został przerwany w czasie uśpienia.
  }
  finally
  {
    Czyszczenie w razie potrzeby
  }
  // Wyjście z metody run powoduje zamknięcie wątku.
}
```

Istnieją dwie bardzo podobne do siebie metody: `interrupted` oraz `isInterrupted`. Pierwsza z nich jest statyczna i sprawdza, czy *bieżący* wątek nie został przerwany. Dodatkowo jej wywołanie powoduje *wyzerowanie* statusu przerwania wątku. Druga natomiast jest metodą obiektową, za pomocą której można sprawdzić status przerwania dowolnego wątku, bez jego zmiany.

Często spotyka się fragmenty kodu, w których wyjątek `InterruptedException` jest tłumiony w następujący sposób:

```
void mySubTask()
{
  . . .
  try { sleep(delay); }
  catch (InterruptedException e) {}    // Nie ignoruj!
  . . .
}
```

Nie należy tego robić! Jeśli nie masz żadnego dobrego pomysłu na klauzulę `catch`, masz jeszcze dwa inne dobre wyjścia:

- W klauzuli `catch` umieść instrukcję `Thread.currentThread().interrupt()`, która ustawi status przerwania. Dzięki temu wywołujący będzie mógł sprawdzić wyjątek.

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

- Lepszym wyjściem jest dodanie do metody instrukcji `throws InterruptedException` i pominięcie bloku `try`. Wtedy wywołujący (lub ostatecznie metoda `run`) może przechwycić ten wyjątek.

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

`java.lang.Thread` **1.0**

- `void interrupt()`

  Wysyła żądanie przerwania do wątku. Status przerwania zostaje ustawiony na `true`. Jeśli wątek jest aktualnie zablokowany przez metodę `sleep`, zgłaszany jest wyjątek `InterruptedException`.

- `static boolean interrupted()`

  Sprawdza, czy *bieżący* wątek (to znaczy ten, który wykonuje tę instrukcję) nie został przerwany. Należy zauważyć, że jest to metoda statyczna. Jej wywołanie ma jeden efekt uboczny — zeruje status przerwania bieżącego wątku na wartość `false`.

- `boolean isInterrupted()`

  Sprawdza, czy wątek nie został przerwany. W przeciwieństwie do statycznej metody `interrupted`, ta nie zmienia statusu przerwania wątku.

- `static Thread currentThread()`

  Zwraca obiekt `Thread` reprezentujący aktualnie wykonywany wątek.

12.3.2. Wątki demony

Aby zamienić zwykły wątek w **demona**, należy użyć poniższej instrukcji:

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

Nie ma w nim jednak nic demonicznego. Demon to taki wątek, którego istnienie polega na służeniu innym wątkom. Należą do nich wątki zegarowe, które wysyłają w równych odstępach czasu tyknięcia zegara do innych wątków, lub takie, które usuwają przestarzałe obiekty z pamięci podręcznej. Jeśli w programie pozostaną same demony, maszyna wirtualna zostaje zamknięta, ponieważ nie ma sensu kontynuować działania programu, w którym nie ma nic poza demonami.

`java.lang.Thread` **1.0**

- `void setDaemon(boolean isDaemon)`

  Określa, czy wątek jest demonem, czy zwykłym wątkiem. Wywołanie tej metody musi nastąpić przed uruchomieniem wątku.

12.3.3. Nazwy wątków

Domyślnie wątkom nadawane są proste nazwy typu `Thread-2`. Można je zmieniać na własne za pomocą metody `setName`:

```
var t = new Thread(runnable);
t.setName("Robot sieciowy");
```

Ułatwia to orientację w zrzutach wątków.

12.3.4. Procedury obsługi nieprzechwyconych wyjątków

Metoda `run` wątku nie może zgłaszać wyjątków kontrolowanych, ale jej działanie może zakończyć wyjątek niekontrolowany. W takiej sytuacji wątek zostaje zamknięty.

Nie ma jednak klauzuli `catch`, do której wyjątek taki można by było przesłać. Zamiast tego bezpośrednio przed zamknięciem wątku wyjątek jest przekazywany do procedury obsługi nieprzechwyconych wyjątków.

Obiekt ten musi należeć do klasy implementującej interfejs `Thread.UncaughtExceptionHandler`. Interfejs ten posiada jedną metodę:

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

Procedurę obsługi w wątku można zainstalować za pomocą metody `setUncaughtExceptionHandler`. Można także dodać procedurę domyślną dla wszystkich wątków za pomocą statycznej metody `setDefaultUncaughtExceptionHandler` z klasy `Thread`. Taka zapasowa procedura mogłaby za pośrednictwem API rejestracyjnego wysyłać raporty o nieprzechwyconych wyjątkach do dziennika.

Jeśli domyślna procedura obsługi nie zostanie zainstalowana, będzie ona `null`. Jeśli jednak zabraknie procedury obsługi dla konkretnego wątku, będzie nią jego obiekt typu `ThreadGroup`.

Grupa to kolekcja wątków, którymi można zarządzać razem. Domyślnie wszystkie tworzone wątki należą do tej samej grupy, ale można założyć inne zgrupowania. Ponieważ teraz są lepsze narzędzia do operowania kolekcjami wątków, nie należy w programach używać grup.

Klasa `ThreadGroup` implementuje interfejs `Thread.UncaughtExceptionHandler`. Znajdująca się w nim metoda `uncaughtException` wykonuje następujące działania:

1. Jeśli grupa wątków posiada rodzica, wywoływana jest metoda `uncaughtException` grupy nadrzędnej.
2. W przeciwnym razie, jeśli metoda `Thread.getDefaultExceptionHandler` zwraca procedurę obsługi niebędącą `null`, metoda `uncaughtException` zostaje wywołana.
3. W przeciwnym razie, jeśli `Throwable` jest egzemplarzem klasy `ThreadDeath`, nic się nie dzieje.
4. W przeciwnym razie nazwa wątku i dane ze śledzenia stosu zostają wydrukowane w strumieniu `System.err`.

Dane ze śledzenia stosu z pewnością każdy widział już wiele razy w swoich programach.

`java.lang.Thread` **1.0**

- `static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` **5.0**
- `static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()` **5.0**

  Ustawia lub zwraca domyślną procedurę obsługi dla nieprzechwyconych wyjątków.
- `void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` **5.0**
- `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()` **5.0**

  Ustawia lub zwraca procedurę obsługi nieprzechwyconych wyjątków. Jeśli nie ma zainstalowanej procedury, jej funkcję pełni obiekt grupy wątków.

`java.lang.Thread.UncaughtExceptionHandler` **5.0**

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

  Zapisuje w dzienniku raport, gdy wątek zostanie zamknięty z powodu nieprzechwyconego wyjątku.

`java.lang.ThreadGroup` **1.0**

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

  Wywołuje metodę nadrzędnej grupy wątków, jeśli taka istnieje, lub domyślną procedurę obsługi klasy `Thread`, jeśli istnieje domyślna procedura, lub w przeciwnym przypadku drukuje dane ze śledzenia stosu w standardowym strumieniu błędów (jeśli `e` jest obiektem typu `ThreadDeath`, dane ze śledzenia stosu są tłumione; obiekty `ThreadDeath` są generowane przez odradzaną metodę `stop`).

12.3.5. Priorytety wątków

Każdy wątek ma swój **priorytet** (ang. *priority*). Domyślnie wątek dziedziczy priorytet po wątku, w którym został utworzony. Aby zmniejszyć lub zwiększyć priorytet wątku, należy użyć metody `setPriority`. Priorytet wątku może mieć dowolną wartość z przedziału od `MIN_PRIORITY` (`1` w klasie `Thread`) do `MAX_PRIORITY` (`10` w klasie `Thread`). Priorytet normalny (`NORM_PRIORITY`) ma wartość `5`.

Kiedy algorytm planujący musi wybrać nowy wątek, decyduje się na ten o najwyższym priorytecie. Należy jednak pamiętać, że priorytety wątków są w dużym stopniu *uzależnione od systemu*. Jeśli maszyna wirtualna korzysta z implementacji wątków leżącej u podłoża platformy, priorytety Javy są odwzorowywane na poziomy platformy, która może dysponować większą lub mniejszą liczbą poziomów priorytetu.

Na przykład system Windows wyróżnia siedem poziomów priorytetu, a więc niektóre z priorytetów Javy zostaną odwzorowane na tym samym poziomie priorytetów systemowych. W maszynie wirtualnej Javy firmy Oracle dla systemu Linux priorytety wątków są całkiem ignorowane — wszystkie wątki mają taki sam priorytet.

Priorytety wątków miały praktyczne zastosowanie we wczesnych wersjach Javy, które nie używały wątków systemu operacyjnego. Dziś nie powinno się już ich używać.

`java.lang.Thread` **1.0**

- `void setPriority(int newPriority)`

  Ustawia priorytet wątku. Wartość priorytetu musi się mieścić w przedziale od `Thread.MIN_PRIORITY` do `Thread.MAX_PRIORITY`. Normalny priorytet określa wartość `Thread.NORM_PRIORITY`.

- `static int MIN_PRIORITY`

  Najmniejszy priorytet, jaki może mieć wątek. Wartość minimalnego priorytetu to `1`.

- `static int NORM_PRIORITY`

  Domyślny priorytet — domyślnie jest to wartość `5`.

- `static int MAX_PRIORITY`

  Najwyższy priorytet, jaki może mieć wątek. Maksymalna wartość priorytetu to `10`.

12.4. Synchronizacja

W większości aplikacji wielowątkowych znajdujących praktyczne zastosowanie jeden zestaw danych jest współdzielony przez co najmniej dwa wątki. Co się stanie, jeśli dwa wątki mające dostęp do tego samego obiektu wywołają metodę zmieniającą jego stan? Jak pewnie się domyślasz, wątki mogą sobie wzajemnie przeszkadzać. Zależnie od kolejności dostępu do danych, w opisanych wyżej sytuacjach mogą powstawać uszkodzone obiekty. Sytuacje te nazywa się **wyścigami** (ang. *race condition*).

12.4.1. Przykład sytuacji powodującej wyścig

Aby uniknąć uszkodzenia współdzielonych przez wątki danych, trzeba umieć *synchronizować dostęp* do nich. W tym podrozdziale zobaczysz, co się dzieje, jeśli zabraknie synchronizacji. W kolejnym natomiast nauczysz się synchronizować operacje dostępu do danych.

W kolejnym przykładzie kontynuujemy naszą symulację banku. W odróżnieniu od przykładu pokazanego w podrozdziale 12.1, „Czym są wątki”, w tym przypadku będziemy losowo wybierać źródło i cel przelewu. Ponieważ będą z tym problemy, dokładniej przyjrzymy się metodzie `transfer` klasy `Bank`.

```
public void transfer(int from, int to, double amount)
  // Ostrzeżenie: metoda niebezpieczna, jeśli wywoływana w kilku wątkach.
{
  System.out.print(Thread.currentThread());
  accounts[from] -= amount;
  System.out.printf(" %10.2f z %d na %d", amount, from, to);
  accounts[to] += amount;
  System.out.printf(" Saldo ogólne: %10.2f%n", getTotalBalance());
}
```

Poniżej znajduje się kod klasy `Runnable`. Jej metoda `run` przelewa pieniądze z określonego konta bankowego. W każdej iteracji metoda ta losowo wybiera jedno konto docelowe i sumę pieniędzy do przelania, wywołuje metodę `transfer` na rzecz obiektu banku i przechodzi w stan uśpienia.

```
Runnable r = () -> {
   try
   {
      while (true)
      {
         int toAccount = (int) (bank.size() * Math.random());
         double amount = MAX_AMOUNT * Math.random();
         bank.transfer(fromAccount, toAccount, amount);
         Thread.sleep((int) (DELAY * Math.random()));
      }
   }
   catch (InterruptedException e)
   {
   }
};
```

W żadnym momencie działania symulacji nie wiadomo, ile jest pieniędzy na każdym z kont. Wiadomo natomiast, że ogólna suma nie powinna się zmieniać, ponieważ program tylko przelewa środki pomiędzy różnymi kontami.

Na końcu każdej transakcji metoda `transfer` oblicza sumę pieniędzy dostępnych na wszystkich kontach i drukuje wynik.

Program ten nigdy się nie kończy. Aby go zamknąć, należy nacisnąć kombinację klawiszy *Ctrl+C*.

Oto typowy wydruk z programu:

```
. . .
Thread[Thread-11,5,main] 588.48 z 11 na 44 Saldo ogólne: 100000.00
Thread[Thread-12,5,main] 976.11 z 12 na 22 Saldo ogólne: 100000.00
Thread[Thread-14,5,main] 521.51 z 14 na 22 Saldo ogólne: 100000.00
Thread[Thread-13,5,main] 359.89 z 13 na 81 Saldo ogólne: 100000.00
. . .
Thread[Thread-36,5,main] 401.71 z 36 na 73 Saldo ogólne: 99291.06
Thread[Thread-35,5,main] 691.46 z 35 na 77 Saldo ogólne: 99291.06
Thread[Thread-37,5,main] 78.64 z 37 na 3 Saldo ogólne: 99291.06
Thread[Thread-34,5,main] 197.11 z 34 na 69 Saldo ogólne: 99291.06
Thread[Thread-36,5,main] 85.96 z 36 na 4 Saldo ogólne: 99291.06
. . .
Thread[Thread-4,5,main]Thread[Thread-33,5,main] 7.31 z 31 na 32 Saldo ogólne:
99979.24
627.50 z 4 na 5 Saldo ogólne: 99979.24
. . .
```

Jak widać, program zawiera poważny błąd. Przez kilka transakcji saldo łączne wszystkich rachunków wynosi 100 000 dolarów, co jest prawidłową kwotą, zważywszy, że jest 100 kont po 1000 dolarów. Jednak po jakimś czasie saldo ulega nieznacznej zmianie. Błędy w obliczeniach mogą się pojawić na krótko po uruchomieniu programu lub dopiero po dłuższym czasie. Taka sytuacja nie napawa optymizmem i z pewnością nikt nie chciałby złożyć w tym banku swoich ciężko zarobionych pieniędzy.

Spróbuj znaleźć błąd w kodzie na listingu 12.3, a rozwiązanie zagadki znajdziesz w kolejnej sekcji.

Listing 12.3. unsynch/UnsynchBankTest.java

```
package unsynch;

/**
 * Program demonstrujący zniszczenie danych spowodowane dostępem kilku wątków do struktury danych.
 * @version 1.32 2018-04-10
 * @author Cay Horstmann
 */
public class UnsynchBankTest
{
   public static final int NACCOUNTS = 100;
   public static final double INITIAL_BALANCE = 1000;
   public static final double MAX_AMOUNT = 1000;
   public static final int DELAY = 10;
```

```
      public static void main(String[] args)
      {
         var bank = new Bank(NACCOUNTS, INITIAL_BALANCE);
         for (int i = 0; i < NACCOUNTS; i++)
         {
            int fromAccount = i;
            Runnable r = () -> {
               try
               {
                  while (true)
                  {
                     int toAccount = (int) (bank.size() * Math.random());
                     double amount = MAX_AMOUNT * Math.random();
                     bank.transfer(fromAccount, toAccount, amount);
                     Thread.sleep((int) (DELAY * Math.random()));
                  }
               }
               catch (InterruptedException e)
               {
               }
            };
            var t = new Thread(r);
            t.start();
         }
      }
   }
```

12.4.2. Wyścigi

W poprzednim podrozdziale napisaliśmy program, w którym kilka wątków aktualizowało salda na kontach bankowych. Po jakimś czasie wkradał się błąd, który powodował pojawienie się lub zniknięcie pewnej kwoty pieniędzy. Problem ten występował w sytuacjach, w których dwa wątki równocześnie próbowały zaktualizować jedno konto. Wyobraźmy sobie, że dwa wątki w tej samej chwili wykonują poniższą instrukcję:

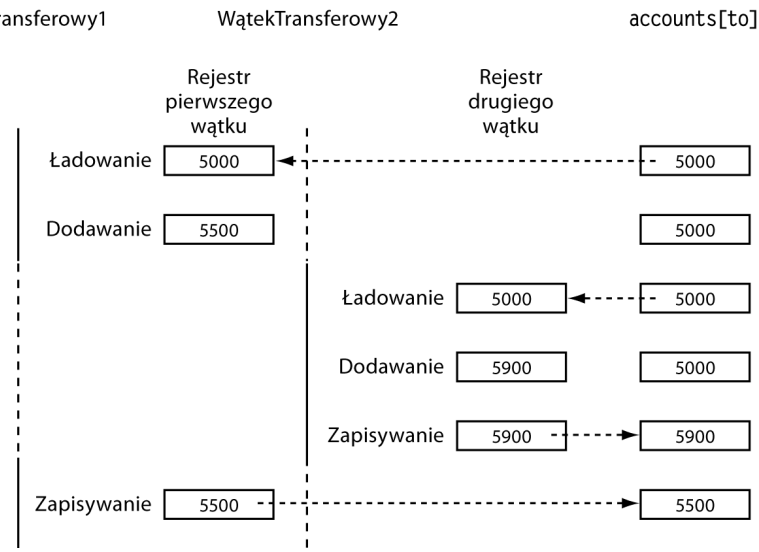
```
accounts[to] += amount;
```

Problem polega na tym, że nie są to operacje **niepodzielne**. Instrukcja ta może zostać wykonana w następujący sposób:

1. Załadowanie `accounts[to]` do rejestru.
2. Dodanie wartości `amount`.
3. Zapisanie wyniku z powrotem w `accounts[to]`.

Wyobraźmy sobie teraz, że pierwszy z wątków wykonuje dwa pierwsze kroki i zostaje wywłaszczony. Następnie budzi się drugi wątek, który aktualizuje tę samą pozycję w tablicy `accounts`. Potem budzi się pierwszy wątek i kończy działanie, wykonując krok trzeci.

Ta czynność wymazuje zmiany dokonane przez drugi wątek, w wyniku czego zmienia się ogólna suma (zobacz rysunek 12.2).



Rysunek 12.2. Jednoczesny dostęp przez dwa wątki

Nasz program testowy wykrywa ten błąd (oczywiście istnieje niewielkie ryzyko fałszywego alarmu, który może nastąpić w sytuacji, gdy zostanie zakłócona praca wątku przeprowadzającego test).

Istnieje możliwość podejrzenia kodu bajtowego maszyny wirtualnej wykonującego każdą z instrukcji klasy. W tym celu należy za pomocą poniższego polecenia zdekompilować plik *Bank.class*:

```
javap -c -v Bank
```

Na przykład kod bajtowy odpowiadający instrukcji `accounts[to] += amount` jest następujący:

```
aload_0
getfield #2;    //Field accounts:[D
iload 2
dup2
daload
dload_3
dadd
dastore
```

Nieważne, co oznaczają te wszystkie instrukcje. Problem polega na tym, że instrukcja zwiększenia wartości została rozbita na kilka mniejszych instrukcji, a praca wykonującego je wątku może zostać przerwana w każdym momencie.

Jakie jest ryzyko wystąpienia tego błędu? W przypadku nowoczesnego procesora wielordzeniowego jest ono dość duże. Zwiększyliśmy szanse na zaobserwowanie problemu na procesorze jednordzeniowym, przeplatając instrukcje drukowania z instrukcjami aktualizującymi saldo.

Jeśli usuniemy instrukcje drukowania, ryzyko znacznie się zmniejszy, ponieważ każdy wątek przed zaśnięciem będzie wykonywał bardzo mało pracy, a poza tym jest mało prawdopodobne, aby algorytm planujący wywłaszczył wątek w trakcie wykonywania obliczeń. Nie znaczy to jednak, że ryzyka wystąpienia błędu nie ma już w ogóle. Jeśli na poważnie obciążonej maszynie

uruchomimy bardzo dużo wątków, program nadal będzie robił błędy i nie pomoże usunięcie instrukcji drukujących. Na wystąpienie błędów może przyjść nam czekać kilka minut, godzin, a nawet dni. Szczerze mówiąc, w życiu programisty jest niewiele gorszych rzeczy od błędu, który daje o sobie znać nieregularnie.

Istotą problemu jest to, że działanie metody `transfer` może zostać przerwane w środku operacji. Gdybyśmy zapewnili ukończenie metody przed utratą przez wątek kontroli, stan obiektu konta bankowego byłby niezagrożony przez błędy.

12.4.3. Obiekty klasy Lock

Do dyspozycji programistów są dwa mechanizmy służące do ochrony bloków kodu przed jednoczesnym dostępem kilku wątków. Służy do tego słowo kluczowe `synchronized`, a w Java SE 5.0 wprowadzono klasę `ReentrantLock`. Słowo kluczowe `synchronized` automatycznie zakłada blokadę oraz tworzy odpowiadający jej warunek, dzięki czemu jest bardzo pożytecznym i wygodnym w użyciu narzędziem wykorzystywanym w większości sytuacji, w których potrzebna jest jawna blokada. Wydaje nam się jednak, że działanie tego słowa kluczowego łatwiej zrozumieć po zapoznaniu się z blokadami i warunkami osobno. Klasy implementujące te podstawowe funkcje znajdują się w pakiecie `java.util.concurrent`. Opisujemy je poniżej i w punkcie 12.4.4, „Warunki”. Po zapoznaniu się z tymi podstawowymi elementami przejdziemy do punktu 12.4.5, „Słowo kluczowe synchronized”.

Szkielet konstrukcji chroniącej blok kodu przy użyciu klasy `ReentrantLock` ma następującą postać:

```
myLock.lock();          // Obiekt klasy ReentrantLock.
try
{
  sekcja krytyczna
}
finally
{
  myLock.unlock();      // Zapewnienie, że blokada zostanie zdjęta, nawet jeśli wystąpi wyjątek.
}
```

Dostęp do sekcji krytycznej powyższej instrukcji w jednym czasie może mieć tylko jeden wątek. Kiedy jeden wątek zablokuje obiekt blokady, żaden inny wątek nie będzie mógł przejść przez instrukcję `lock`. Jeśli jakiś inny wątek wywoła metodę `lock`, zostanie dezaktywowany do czasu, aż poprzedni wątek odblokuje obiekt blokady.

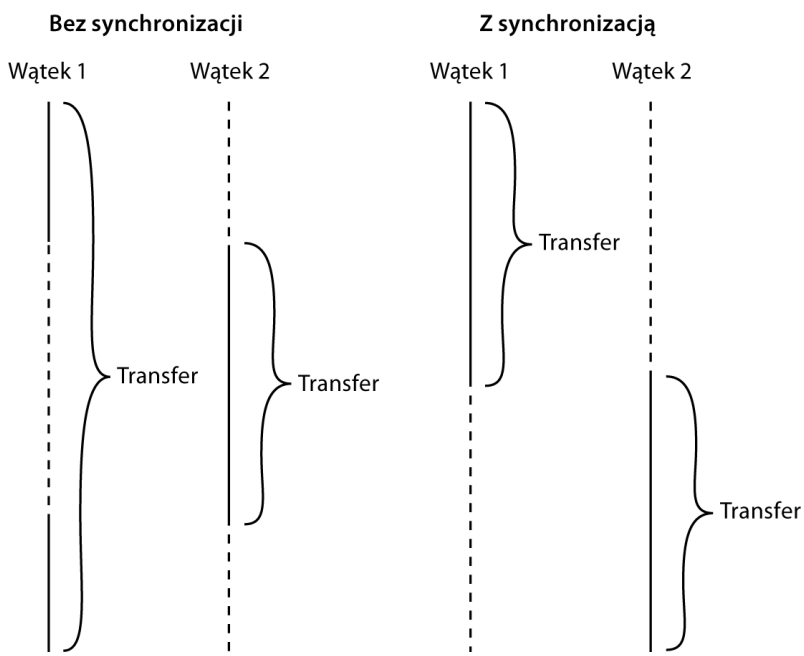
Metoda `unlock` musi się bezwzględnie znajdować w bloku `finally`. Jeśli kod w sekcji krytycznej spowoduje wyjątek, blokada musi zostać zdjęta. W przeciwnym przypadku reszta wątków pozostanie zablokowana na zawsze.

Z blokadami nie można używać instrukcji `try` z zasobami. Przede wszystkim metoda zdejmująca blokadę nie nazywa się `close`. Jednak nawet gdyby zmieniono jej nazwę, to instrukcja `try` z zasobami i tak by nie działała. W jej nagłówku powinna się znaleźć deklaracja nowej zmiennej, a w blokadzie używa się jednej zmiennej, z której korzystają różne wątki.

Spróbujmy za pomocą blokady ochronić metodę `transfer` z klasy `Bank`.

```
public class Bank
{
   private var bankLock = new ReentrantLock();    // Klasa ReentrantLock implementuje
                                                  // interfejs Lock.
   . . .
  public void transfer(int from, int to, int amount)
  {
    bankLock.lock();
    try
    {
      System.out.print(Thread.currentThread());
      accounts[from] -= amount;
      System.out.printf(" %10.2f z %d na %d", amount, from, to);
      accounts[to] += amount;
      System.out.printf(" Saldo ogólne: %10.2f%n", getTotalBalance());
    }
    finally
    {
      bankLock.unlock();
    }
  }
}
```

Załóżmy, że jakiś wątek wywołuje metodę `transfer`, ale zostaje wywłaszczony przed jej ukończeniem. Następnie inny wątek również wywołuje tę metodę. Nie może on jednak założyć blokady i zostaje zablokowany wywołaniem metody `lock`. Jest dezaktywowany i musi poczekać, aż pierwszy wątek skończy wykonywanie metody `transfer`. Kiedy ten zdejmie blokadę, drugi wątek może kontynuować (zobacz rysunek 12.3).



Rysunek 12.3. Porównanie wątków synchronizowanych i niesynchronizowanych

Wypróbuj, czy to działa. Dodaj kod blokujący do metody `transfer` i ponownie uruchom program, a przekonasz się, że saldo bankowe nie zmieni się bez względu na długość czasu działania programu.

Należy zauważyć, że każdy obiekt klasy `Bank` posiada własny obiekt klasy `ReentrantLock`. Jeśli dwa wątki próbują uzyskać dostęp do tego samego obiektu `Bank`, blokada ustawia je w kolejce. Jeśli natomiast każdy z wątków dobiera się do innego obiektu `Bank`, zakładają one osobne blokady i żaden z nich nie jest blokowany. Jest to jak najbardziej prawidłowe działanie, ponieważ wątki działające na różnych obiektach nie mogą sobie przeszkadzać.

Blokada ta jest **wielowejściowa** (ang. *reentrant*), ponieważ wątek może wielokrotnie zakładać blokadę, którą już posiada. Blokada posiada **licznik pamiętający** liczbę zagnieżdżonych wywołań metody `lock`. Dlatego, aby blokada została zwolniona, wątek musi wywołać tyle razy metodę `unlock`, ile razy wywołał metodę `lock`. Dzięki temu kod chroniony przez blokadę może wywołać inną metodę, która wykorzystuje te same blokady.

Na przykład metoda `transfer` wywołuje metodę `getTotalBalance`, która blokuje obiekt `bankLock` mający obecnie licznik o wartości `2`. Kiedy metoda `getTotalBalance` kończy działanie, wartość licznika spada do `1`. Zakończenie metody `transfer` zmniejsza go do `0` i wątek zwalnia blokadę.

Z reguły ochroną obejmuje się bloki kodu, które aktualizują lub badają współdzielone obiekty. Daje to pewność, że operacja zostanie zakończona, zanim inny wątek będzie mógł użyć tego samego obiektu.

Trzeba uważać, aby procedury zawarte w sekcji krytycznej nie zostały pominięte z powodu wystąpienia wyjątku. Jeśli wyjątek wystąpi przed końcem sekcji, klauzula `finally` zwolni blokadę, ale obiekt może pozostać w naruszonym stanie.

`java.util.concurrent.Locks.Lock` **5.0**

- `void lock()`

  Zakłada blokadę. Zostaje zablokowana, jeśli blokada ta jest aktualnie w posiadaniu innego wątku.

- `void unlock()`

  Zwalnia blokadę.

`java.util.concurrent.locks.ReentrantLock` **5.0**

- `ReentrantLock()`

  Tworzy wielowejściową blokadę, za pomocą której można chronić sekcję krytyczną.

- `ReentrantLock(boolean fair)`

  Tworzy obiekt blokady z określoną zasadą uczciwości. Uczciwa blokada ustawia na pierwszym miejscu wątek, który czekał najdłużej. Jednak zasada ta może powodować duże straty szybkości. Dlatego domyślnie blokady nie muszą być uczciwe.

Opcja uczciwości wydaje się lepszym rozwiązaniem, ale uczciwe blokady są *znacznie wolniejsze* od zwykłych. Uczciwe blokady należy stosować wyłącznie w sytuacjach, w których takie zachowanie jest całkowicie niezbędne. Stosując uczciwą blokadę, nie ma gwarancji, że algorytm odpowiedzialny za harmonogram uruchamiania wątków również jest uczciwy. Jeśli algorytm ten dyskryminuje wątek, który oczekiwał przez długi czas na blokadę, blokada nie ma szansy potraktować go lepiej.

12.4.4. Warunki

Często zdarza się tak, że po wejściu do sekcji krytycznej wątek dowiaduje się, iż nie może kontynuować, dopóki nie zostanie spełniony warunek. Do zarządzania wątkami, które uzyskały blokadę, ale nie mogą robić nic pożytecznego, służą **obiekty warunków**. W tym rozdziale opisujemy implementację warunków w bibliotece Javy (ze względu na przeszłość obiekty warunków są czasami nazywane **zmiennymi warunkowymi**).

Ulepszymy naszą symulację banku. Nie chcemy, aby pieniądze były przelewane z kont, na których nie ma wystarczających środków. Zauważ, że nie możemy użyć instrukcji jak poniżej:

```
if (bank.getBalance(from) >= amount)
bank.transfer(from, to, amount);
```

Jest całkiem możliwe, że aktualny wątek zostanie dezaktywowany pomiędzy pomyślnym wynikiem testu a wywołaniem metody `transfer`.

```
if (bank.getBalance(from) >= amount)
// W tym miejscu wątek może być nieaktywny.
bank.transfer(from, to, amount);
```

Zanim wątek zostanie ponownie uruchomiony, saldo na koncie może spaść poniżej minimalnej potrzebnej kwoty. Należy przypilnować, aby żaden wątek nie zmodyfikował salda pomiędzy testem a wykonaniem przelewu. Dlatego zarówno test, jak i operację przelewu chronimy przy użyciu blokady:

```
public void transfer(int from, int to, int amount)
{
  bankLock.lock();
  try
  {
    while (accounts[from] < amount)
    {
      // czekanie
      . . .
    }
    // przelew środków
    . . .
  }
  finally
  {
    bankLock.unlock();
  }
}
```

Kolej na podjęcie decyzji, co zrobić, jeśli na koncie będzie za mało pieniędzy. W takiej sytuacji czekamy, aż jakiś inny wątek zwiększy jego saldo. Pamiętamy jednak, że pierwszy wątek całkowicie zablokował dostęp do obiektu `bankLock`, przez co żaden inny wątek nie może dokonać depozytu. W takim przypadku do gry wchodzą obiekty warunków.

Z obiektem blokady może być związanych nawet kilka warunków. Obiekty warunków tworzy się za pomocą metody `newCondition`. Istnieje zwyczaj nadawania obiektom warunków takich nazw, które w jakiś sposób przypominają reprezentowane warunki. Na przykład w poniższym fragmencie programu tworzymy obiekt warunku reprezentujący warunek wystarczających środków.

```
class Bank
{
  private Condition sufficientFunds;
  . . .
  public Bank()
  {
    . . .
    sufficientFunds = bankLock.newCondition();
  }
}
```

Jeśli metoda `transfer` odkryje, że na koncie nie ma dostępnych wystarczających środków, wykonuje instrukcję `sufficientFunds.await();`.

Dzięki temu aktualny wątek zostaje dezaktywowany i następuje zdjęcie blokady. To umożliwia działanie kolejnemu wątkowi, który, mamy nadzieję, zwiększy saldo konta.

Pomiędzy wątkiem, który oczekuje na blokadę, a wątkiem, który wywołał metodę `await`, istnieje zasadnicza różnica. Ten drugi zostaje umieszczony w kolejce **wątków oczekujących** (ang. *wait set*) warunku. Wątek ten *nie* przechodzi w stan wykonywalności, dopóki inny wątek nie wywoła metody `signalAll` na rzecz tego samego warunku.

Inny wątek przelewający pieniądze powinien wykonać instrukcję `sufficientFunds.signal`
`↪All();`.

Powoduje ona reaktywację wszystkich wątków oczekujących na warunek. Wątki usunięte z kolejki oczekujących są z powrotem wykonywalne, a algorytm odpowiedzialny za harmonogram w końcu ponownie je uaktywni. Wtedy spróbują one ponownie wejść do obiektu. Jak tylko będzie dostępna blokada, jeden z wątków ją założy i będzie kontynuował pracę *od momentu, w którym ją przerwał*, wracając z wywołania metody `await`.

W tym momencie wątek powinien ponownie sprawdzić warunek. Nie ma gwarancji, że teraz zostanie on spełniony. Metoda `signalAll` tylko sygnalizuje wątkom, że tym razem warunek *może* zostać spełniony i dlatego dobrze by było to sprawdzić.

Ogólnie rzecz biorąc, metoda `await` powinna być wywoływana w pętli o następującej formie:

```
while (!(można kontynuować))
  condition.await();
```

Ważne jest, aby metoda `signalAll` była wywoływana także przez *jakiś* inny wątek, ponieważ wątek wywołujący metodę `await` nie ma możliwości reaktywowania samego siebie. Musi on liczyć na inne wątki. Jeśli żaden z nich go nie reaktywuje, nie zostanie on nigdy więcej uruchomiony. To może prowadzić do nieprzyjemnych **zakleszczeń** (ang. *deadlock*). Jeśli prawie wszystkie wątki zostaną zablokowane, a ostatni aktywny wątek wywoła metodę `await`, nie odblokowując reszty, nie będzie komu zdjąć blokady i program zawiesi się.

Kiedy powinno się wywoływać metodę `signalAll`? Główna reguła nakazuje zrobienie tego zawsze wtedy, gdy stan obiektu zmieni się w taki sposób, który może być korzystny dla wątków oczekujących. Na przykład wątki powinny mieć możliwość sprawdzenia salda na koncie za każdym razem, gdy ulegnie ono zmianie. W naszym przykładowym programie metodę `signalAll` wywołujemy po zakończeniu przelewu pieniędzy.

```
public void transfer(int from, int to, int amount)
{
  bankLock.lock();
  try
  {
    while (accounts[from] < amount)
    sufficientFunds.await();
    // przelew środków
    . . .
    sufficientFunds.signalAll();
  }
  finally
  {
    bankLock.unlock();
  }
}
```

Należy pamiętać, że wywołanie metody `signalAll` nie powoduje natychmiastowej aktywacji oczekującego wątku. Ona tylko odblokowuje oczekujące wątki, aby mogły konkurować o wejście do obiektu po tym, jak bieżący wątek zwolni blokadę.

Istnieje także metoda `signal`, która odblokowuje tylko jeden losowo wybrany wątek. Jest to mniej obciążająca czynność niż odblokowywanie wszystkich wątków, ale wiąże się z nią pewne ryzyko. Jeśli losowo wybrany wątek „dojdzie do wniosku”, że nadal nie może nic zrobić, zostanie z powrotem zablokowany. Jeśli żaden inny wątek nie wywoła metody `signal` jeszcze jeden raz, system ulegnie zakleszczeniu.

Wątek pozostający w posiadaniu blokady warunku może na jego rzecz wywołać tylko metodę `await`, `signalAll` lub `signal`.

Po uruchomieniu programu przedstawionego na listingu 12.4 widać, że nie ma żadnych błędów w obliczeniach. Saldo ogólne cały czas wynosi 1000 dolarów. Saldo żadnego z kont nigdy nie jest ujemne (przypominamy, że aby zakończyć program, trzeba wcisnąć kombinację klawiszy *Ctrl+C*). Da się również zauważyć, że program działa nieco wolniej — jest to cena, jaką płacimy za synchronizację.

Listing 12.4. synch/Bank.java

```
package synch;

import java.util.concurrent.locks.*;

/**
 * Bank z kilkoma kontami, kontrolujący dostęp za pomocą blokad
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
 */
public class Bank
{
   private final double[] accounts;
   private Lock bankLock;
   private Condition sufficientFunds;

   /**
    * Tworzy bank
    * @param n liczba kont
    * @param initialBalance saldo początkowe na każdym koncie
    */
   public Bank(int n, double initialBalance)
   {
      accounts = new double[n];
      Arrays.fill(accounts, initialBalance);
      bankLock = new ReentrantLock();
      sufficientFunds = bankLock.newCondition();
   }

   /**
    * Przelewa pieniądze pomiędzy kontami.
    * @param from konto, z którego ma nastąpić przelew
    * @param to konto, na które mają zostać przelane środki
    * @param amount kwota do przelania
    */
   public void transfer(int from, int to, double amount) throws InterruptedException
   {
      bankLock.lock();
      try
      {
         while (accounts[from] < amount)
            sufficientFunds.await();
         System.out.print(Thread.currentThread());
         accounts[from] -= amount;
         System.out.printf(" %10.2f z %d na %d", amount, from, to);
         accounts[to] += amount;
         System.out.printf(" Saldo ogólne: %10.2f%n", getTotalBalance());
         sufficientFunds.signalAll();
      }
      finally
      {
         bankLock.unlock();
      }
   }

   /**
```

```
      * Zwraca sumę sald wszystkich kont.
      * @return saldo ogólne
      */
     public double getTotalBalance()
     {
        bankLock.lock();
        try
        {
           double sum = 0;

           for (double a : accounts)
              sum += a;

           return sum;
        }
        finally
        {
           bankLock.unlock();
        }
     }

     /**
      * Zwraca liczbę kont w banku.
      * @return liczba kont
      */
     public int size()
     {
        return accounts.length;
     }
  }
```

Poprawne zastosowanie warunków w praktyce może być sporym wyzwaniem. Przed podjęciem próby zaimplementowania własnych obiektów warunków dobrze by było najpierw wziąć pod uwagę jedną z konstrukcji opisanych w podrozdziale 12.5, „Kolekcje bezpieczne wątkowo”.

`java.util.concurrent.locks.Lock` **5.0**

- `Condition newCondition()`

  Zwraca obiekt warunku związany z blokadą.

`java.util.concurrent.locks.Condition` **5.0**

- `void await()`

  Umieszcza wątek w kolejce oczekujących do warunku.

- `void signalAll()`

  Odblokowuje wszystkie wątki znajdujące się w kolejce oczekujących do warunku.

- `void signal()`

  Odblokowuje losowo wybrany wątek znajdujący się w kolejce oczekujących do warunku.

12.4.5. Słowo kluczowe synchronized

W poprzednich podrozdziałach nauczyliśmy się używać obiektów typu `Lock` i `Condition`. Zanim przejdziemy dalej, zrobimy podsumowanie najważniejszych wiadomości na temat blokad i warunków:

- Blokada chroni blok kodu, pozwalając wykonywać go tylko jednemu wątkowi w danym czasie.
- Blokada zarządza wątkami, które próbują wejść do chronionego segmentu kodu.
- Z blokadą może być związany jeden lub więcej obiektów warunkowych.
- Każdy obiekt warunkowy zarządza wątkami, które weszły do sekcji kodu chronionego, ale które nie mogą kontynuować działania.

Interfejsy `Lock` i `Condition` umożliwiają programistom zyskanie większej kontroli nad blokadami. W większości sytuacji kontrola ta jest jednak zbędna, ponieważ można wykorzystać mechanizm wbudowany w język. Od wersji 1.0 każdy obiekt w Javie posiada *blokadę wewnętrzną*. Jeśli w deklaracji metody zostanie użyte słowo kluczowe `synchronized`, blokada obiektu chroni całą tę metodę. To znaczy, że aby ją wywołać, wątek musi założyć wewnętrzną blokadę obiektu.

Innymi słowy, poniższy kod:

```
public synchronized void metoda()
{
  ciało metody
}
```

jest równoważny z tym:

```
public void metoda()
{
  this.intrinsicLock.lock();
  try
  {
    ciało metody
  }
  finally { this.intrinsicLock.unlock(); }
}
```

Na przykład zamiast stosować blokadę jawną, wystarczy zadeklarować metodę `transfer` z klasy `Bank` jako synchronizowaną (`synchronized`).

Z wewnętrzną blokadą obiektu jest związany jeden warunek. Metoda `wait` dodaje wątek do kolejki oczekujących, a metody `notifyAll` i `notify` odblokowują oczekujące wątki. Innymi słowy, wywołanie metody `wait` lub `notifyAll` jest równoznaczne z poniższym:

```
intrinsicCondition.await();
intrinsicCondition.signalAll();
```

Metody `wait`, `notifyAll` i `notify` są metodami finalnymi klasy `Object`. Aby uniknąć konfliktów nazw, odpowiadające im metody w interfejsie `Condition` zostały nazwane `await`, `signalAll` i `signal`.

Na przykład implementacja klasy `Bank` może wyglądać następująco:

```
class Bank
{
  private double[] accounts;

  public synchronized void transfer(int from, int to, int amount) throws
InterruptedException
  {
    while (accounts[from] < amount)
      wait();       // Oczekiwanie na warunek wewnętrznej blokady obiektu.
    accounts[from] -= amount;
    accounts[to] += amount;
    notifyAll();    // Powiadomienie wszystkich wątków oczekujących na warunek.
  }
  public synchronized double getTotalBalance() { . . . }
}
```

Jak widać, słowo kluczowe `synchronized` pozwala na pisanie znacznie bardziej zwięzłego kodu. Oczywiście, aby go zrozumieć, trzeba wiedzieć, że każdy obiekt posiada wewnętrzną blokadę, która z kolei posiada wewnętrzny warunek. Blokada zarządza wątkami, które próbują wejść do metody synchronizowanej. Warunek zajmuje się wątkami, które wywołały metodę `wait`.

Metody synchronizowane są względnie proste. Jednak początkujący często szarpią się z warunkami. Przed przejściem do używania metod `wait` i `notifyAll` lepiej zastanowić się nad użyciem jednej z konstrukcji opisanych w podrozdziale 12.5, „Kolekcje bezpieczne wątkowo”.

Synchronizowane mogą być także metody statyczne. Jeśli taka metoda zostanie wywołana, uzyskuje dostęp do blokady wewnętrznej obiektu związanej z nią klasy. Jeśli na przykład klasa `Bank` zawierałaby statyczną metodę synchronizowaną, blokada obiektu `Bank.class` byłaby blokowana w chwili wywołania tej metody. W wyniku tego żaden inny wątek nie mógłby wywołać tej ani żadnej innej statycznej metody synchronizowanej tej klasy.

Wewnętrzne blokady i warunki mają pewne ograniczenia. Oto niektóre z nich:

- Nie można przerwać wątku, który próbuje założyć blokadę.
- Nie można określić maksymalnego czasu próby dostępu do blokady.
- Sytuacja, w której na jedną blokadę przypada jeden warunek, może nie być najlepsza pod kątem wydajności.

Czego najlepiej używać — obiektów `Lock` i `Condition` czy metod synchronizowanych? Oto nasze zalecenia w tej kwestii:

- Najlepiej nie używać interfejsów `Lock` i `Condition` ani słowa kluczowego `synchronized`. W wielu sytuacjach można poradzić sobie przy użyciu jednego z mechanizmów z pakietu `java.util.concurrent`, które zajmują się działaniami związanymi z blokowaniem. Na przykład w punkcie 12.5.1, „Kolejki blokujące”, opisujemy sposób synchronizacji wątków pracujących nad wspólnym zadaniem za pomocą blokowania kolejek. Dobrze jest też poczytać o strumieniach równoległych, o których piszemy w rozdziale 1. drugiego tomu.

- Jeśli słowo kluczowe `synchronized` sprawdza się w określonej sytuacji, należy go użyć. Technika ta pozwala na zmniejszenie liczby wierszy kodu i pozostawia mniej okazji do popełnienia błędu. Listing 12.5 przedstawia program symulujący bank zaimplementowany przy użyciu metod synchronizowanych.
- Interfejsów `Lock` i `Condition` należy używać, gdy nie można się obyć bez dodatkowych funkcji, które one udostępniają.

Listing 12.5. synch2/Bank.java

```
package synch2;

/**
 * Bank z kilkoma kontami, wykorzystujący synchronizację
 */
public class Bank
{
   private final double[] accounts;

   /**
    * Tworzy bank.
    * @param n liczba kont
    * @param initialBalance saldo początkowe na każdym koncie
    */

   public Bank(int n, double initialBalance)
   {
      accounts = new double[n];
      Arrays.fill(accounts, initialBalance);
   }

   /**
    * Przelewa pieniądze pomiędzy kontami.
    * @param from konto, z którego ma nastąpić przelew
    * @param to konto, na które mają zostać przelane środki
    * @param amount kwota do przelania
    */

   public synchronized void transfer(int from, int to, double amount) throws
   ➥InterruptedException
   {
      while (accounts[from] < amount)
         wait();
      System.out.print(Thread.currentThread());
      accounts[from] -= amount;
      System.out.printf(" %10.2f z %d na %d", amount, from, to);
      accounts[to] += amount;
      System.out.printf(" Saldo ogólne: %10.2f%n", getTotalBalance());
      notifyAll();
   }

   /**
    * Zwraca sumę sald wszystkich kont.
    * @return saldo ogólne
    */

   public synchronized double getTotalBalance()
```

```
      {
         double sum = 0;

         for (double a : accounts)
            sum += a;

         return sum;
      }

      /**
       * Zwraca liczbę kont w banku.
       * @return liczba kont
       */

      public int size()
      {
         return accounts.length;
      }
   }
```

`java.lang.Object` **1.0**

- `void notifyAll()`

  Odblokowuje wszystkie wątki, które wywołały metodę `wait` na rzecz obiektu. Metodę tę można wywoływać wyłącznie w synchronizowanej metodzie lub synchronizowanym bloku. Powoduje wyjątek `IllegalMonitorStateException`, jeśli aktualny wątek nie jest właścicielem blokady obiektu.

- `void notify()`

  Odblokowuje jeden losowo wybrany wątek spośród tych, które wywołały metodę `wait` na rzecz obiektu. Metodę tę można wywoływać wyłącznie w synchronizowanej metodzie lub synchronizowanym bloku. Powoduje wyjątek `IllegalMonitorStateException`, jeśli aktualny wątek nie jest właścicielem blokady obiektu.

- `void wait()`

  Przestawia wątek w stan oczekiwania, aż nadejdzie odpowiednie powiadomienie. Metodę tę można wywoływać wyłącznie w synchronizowanej metodzie lub synchronizowanym bloku. Powoduje wyjątek `IllegalMonitorStateException`, jeśli aktualny wątek nie jest właścicielem blokady obiektu.

- `void wait(long millis)`
- `void wait(long millis, int nanos)`

  Przestawia wątek w stan oczekiwania, aż nadejdzie odpowiednie powiadomienie lub upłynie określona ilość czasu. Metodę tę można wywoływać wyłącznie w synchronizowanej metodzie lub synchronizowanym bloku. Powoduje wyjątek `IllegalMonitorStateException`, jeśli aktualny wątek nie jest właścicielem blokady obiektu. Liczba nanosekund nie może być większa niż 1 000 000.

12.4.6. Bloki synchronizowane

Jak już wiemy, każdy obiekt w Javie posiada blokadę. Wątek może ją przejąć za pomocą metody synchronizowanej. Istnieje jeszcze jeden sposób na pozyskiwanie blokad, który polega na wejściu do **bloku synchronizowanego**. Wątek, które wejdzie do bloku podobnego do tego poniżej, stanie się właścicielem blokady dla obiektu `obj`.

```
synchronized (obj)          // składnia bloku synchronizowanego
{
  sekcja krytyczna
}
```

Czasami można spotkać blokady tworzone ad hoc, na przykład:

```
public class Bank
{
  private double[] accounts;
  private var lock = new Object();
  . . .
  public void transfer(int from, int to, int amount)
  {
    synchronized (lock)    // blokada utworzona ad hoc
    {
      accounts[from] -= amount;
      accounts[to] += amount;
    }
    System.out.println(. . .);
  }
}
```

W tym przypadku obiekt `lock` został utworzony tylko po to, aby można było użyć blokady, którą posiada każdy obiekt w Javie.

Czasami programiści wykorzystują blokady obiektów do implementacji dodatkowych niepodzielnych operacji. Technika ta nazywa się **blokowaniem po stronie klienta** (ang. *client-side locking*). Weźmy na przykład klasę `Vector` implementującą listy, których metody są synchronizowane. Wyobraźmy sobie, że salda kont w naszym banku zapisaliśmy w liście `Vector<Double>`. Oto naiwna implementacja metody `transfer`:

```
public void transfer(Vector<Double> accounts, int from, int to, int amount)   // błąd
{
  accounts.set(from, accounts.get(from) - amount);
  accounts.set(to, accounts.get(to) + amount);
  System.out.println(. . .);
}
```

Mimo że metody `get` i `set` klasy `Vector` są synchronizowane, nic nam to nie daje. Istnieje możliwość, że wątek zostanie wywłaszczony w metodzie `transfer` po zakończeniu pierwszego wywołania metody `get`. Wtedy inny wątek może w tym samym miejscu zapisać całkiem inną wartość. Można jednak przejąć blokadę:

```
public void transfer(Vector<Double> accounts, int from, int to, int amount)
{
  synchronized (accounts)
  {
    accounts.set(from, accounts.get(from) - amount);
    accounts.set(to, accounts.get(to) + amount);
```

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

Technika ta zdaje egzamin, ale jest w pełni uzależniona od tego, że wszystkie metody modyfikujące w klasie `Vector` mają wewnętrzne blokady. Czy tak jest jednak naprawdę? W dokumentacji tej klasy nic takiego nie napisano. Trzeba bardzo uważnie przestudiować jej kod źródłowy i mieć nadzieję, że w przyszłości nie zostaną wprowadzone mutatory niesynchronizowane. Stanowi to dowód na to, że blokowanie po stronie klienta jest bardzo niepewną techniką, i dlatego ogólnie nie polecamy jej stosowania.

Maszyna wirtualna Javy ma wbudowane mechanizmy wsparcia dla metod synchronizowanych. Bloki synchronizowane są kompilowane na długie sekwencje kodu bajtowego pozwalające na zarządzanie blokadą wewnętrzną.

12.4.7. Monitor

Blokady i warunki stwarzają bardzo duże możliwości, jeśli chodzi o synchronizację wątków, ale są mało obiektowe. Badacze przez wiele lat poszukiwali sposobów na uczynienie wielowątkowości bezpieczną techniką, nie zmuszając jednocześnie programistów do zajmowania się jawnymi blokadami. Jednym z najlepszych rozwiązań w tej dziedzinie są **monitory**, opracowane w latach 70. ubiegłego wieku przez Pera Brincha Hansena i Tony'ego Hoare'a. W Javie monitor ma następujące własności:

- Monitorem jest klasa posiadająca same pola prywatne.
- Z każdym obiektem tej klasy związana jest blokada.
- Wszystkie metody są blokowane przez tę blokadę. Innymi słowy, jeśli klient wykona instrukcję `obj.method()`, to blokada obiektu `obj` zostanie automatycznie założona na początku wywołania metody i zwolniona w chwili zwrócenia przez tę metodę wartości. Ponieważ wszystkie pola są prywatne, mamy pewność, że podczas gdy jeden wątek wykonuje działania na nich, żaden inny wątek nie ma do nich dostępu.
- Z blokadą może być skojarzona dowolna liczba warunków.

Poprzednie wersje monitorów dysponowały tylko jednym warunkiem o dosyć eleganckiej składni. Można było użyć wywołania typu `await accounts[from] >= balance` bez stosowania jawnej zmiennej warunkowej. Badania wykazały jednak, że masowe powtarzanie testów warunków może być bardzo mało wydajne. Problem ten rozwiązują jawne zmienne warunkowe, z których każda zarządza osobnym zestawem wątków.

Projektanci Javy luźno potraktowali adaptację koncepcji monitorów. *Każdy obiekt* w Javie posiada wewnętrzną blokadę i wewnętrzny warunek. Jeśli w deklaracji metody znajduje się słowo kluczowe `synchronized`, działa ona jak metoda monitorowa. Dostęp do zmiennej warunkowej uzyskuje się za pośrednictwem metod `wait`, `notifyAll` i `notify`.

Pomiędzy zwykłym obiektem a monitorem są trzy istotne różnice mające wpływ na bezpieczeństwo wątków:

- Pola nie muszą być prywatne.

- Metody nie muszą być synchronizowane.
- Blokada wewnętrzna jest dostępna dla klientów.

Ten brak poszanowania dla zabezpieczeń rozwścieczył Pera Brincha Hansena. W zjadliwej recenzji na temat wielowątkowości w Javie napisał: „Zdumiewa fakt, że pozbawiony zabezpieczeń paralelizm w Javie jest poważnie traktowany przez programistów, zwłaszcza że od wynalezienia monitorów i powstania języka Concurrent Pascal upłynęło już pół wieku. To nie ma sensu" (*Java's Insecure Parallelism*, „ACM SIGPLAN Notices" 1999, nr 34, s. 38 – 45).

12.4.8. Pola ulotne

Czasami wydaje się, że obciążenie powodowane przez synchronizację jest zbyt duże, jeśli chcemy tylko odczytać lub zapisać jedno czy dwa pola. Co w takiej sytuacji może się nie udać? Niestety nowoczesne procesory i kompilatory stwarzają mnóstwo sytuacji, w których może zostać popełniony błąd.

- Komputery wieloprocesorowe mogą tymczasowo przechowywać wartości w rejestrach lub lokalnych pamięciach podręcznych. W wyniku tego wątki działające na różnych procesorach mogą w tej samej lokalizacji widzieć inne dane!
- Kompilatory mogą zmieniać kolejność instrukcji w celu zoptymalizowania programu. Kompilator nie zmieni kolejności w taki sposób, aby zmienił się sposób działania kodu, ale wyjdzie z założenia, że wartości w pamięci ulegają zmianom tylko w wyniku konkretnych instrukcji w kodzie. Jednak wartość może zostać zmodyfikowana przez inny wątek!

Problemy te nie występują po zastosowaniu blokad do ochrony kodu, do którego mogą mieć dostęp różne wątki. Kompilatory muszą respektować blokady poprzez opróżnianie w razie potrzeby lokalnych pamięci podręcznych i nieprzestawianie instrukcji w nieodpowiedni sposób. Szczegółowe informacje na ten temat można znaleźć w dokumencie *Java Memory Model and Thread Specification* opracowanym przez grupę JSR 133 (*http://www.jcp.org/en/jsr/detail?id=133*). Znaczna część tej specyfikacji jest bardzo skomplikowana i ma czysto techniczny charakter, ale jest kilka jaśniejszych fragmentów. Bardziej przystępny artykuł na ten temat napisany przez Briana Goetza znajduje się pod adresem *http://www.ibm.com/developerworks/library/j-jtp02244*.

Brian Goetz jest autorem „motta synchronizacyjnego": jeśli zapisujesz zmienną, która może następnie zostać odczytana przez inny wątek, albo odczytujesz zmienną, która mogła zostać zapisana przez inny wątek, musisz skorzystać z synchronizacji.

Słowo kluczowe `volatile` (ulotny) umożliwia synchronizację dostępu do pól egzemplarza bez użycia blokad. Jeśli w deklaracji pola znajduje się to słowo kluczowe, kompilator i maszyna wirtualna wiedzą, że może ono być współbieżnie modyfikowane przez inny wątek.

Wyobraźmy sobie na przykład, że pewien obiekt ma znacznik logiczny `done`, który jest ustawiany przez jeden wątek, a sprawdzany przez inny. Zgodnie z wcześniejszymi informacjami wiemy, że można w tej sytuacji użyć blokady:

```
private boolean done;

public synchronized boolean isDone() { return done; }
public synchronized void setDone() { done = true; }
```

Użycie wewnętrznej blokady wydaje się niezbyt dobrym pomysłem. Metody `isDone` i `setDone` mogą zostać zablokowane, jeśli inny wątek zablokuje obiekt. Jeśli istnieje taka obawa, można zastosować osobną blokadę tylko dla tej zmiennej. To jednak zaczyna robić się coraz bardziej kłopotliwe.

Rozsądnym wyjściem z tej sytuacji jest zadeklarowanie pola jako `volatile`:

```
private volatile boolean done;
public boolean isDone() { return done; }
public void setDone() { done = true; }
```

Kompilator wstawi odpowiedni kod pozwalający dopilnować, by zmiana w zmiennej `done` dokonana w jednym wątku była widzialna w innym wątku, który będzie ją odczytywał.

Zmienne ulotne nie zapewniają niepodzielności. Na przykład nie ma gwarancji, że poniższa metoda zmieni wartość pola na przeciwną.

```
public void flipDone() { done = !done; }    // podzielna
```

Nie ma gwarancji, że coś nie zakłóci odczytu, zamiany i zapisu.

12.4.9. Zmienne finalne

Jak wiesz z poprzedniego podrozdziału, nie można bezpiecznie odczytać pola w wielu wątkach, jeśli nie użyje się blokad lub modyfikatora `volatile`.

Jest jeszcze jedna sytuacja, w której można bezpiecznie uzyskać dostęp do wspólnego pola — gdy pole to jest finalne:

```
final var accounts = new HashMap<String, Double>();
```

Inne wątki zobaczą zmienną `accounts`, gdy konstruktor zakończy działanie.

Bez modyfikatora `final` nie byłoby gwarancji, że wątki zobaczą zaktualizowaną wartość zmiennej `accounts` — mogłyby widzieć `null` zamiast utworzonej struktury `HashMap`.

Oczywiście działania na słowniku nie są bezpieczne wątkowo. Jeśli ma ją modyfikować i odczytywać kilka wątków, to nadal konieczna jest synchronizacja.

12.4.10. Zmienne atomowe

Wspólne zmienne można deklarować jako ulotne, pod warunkiem że jedyną wykonywaną operacją będzie przypisanie.

W pakiecie `java.util.concurrent.atomic` znajduje się kilka klas, które zapewniają atomowość innych operacji dzięki wykorzystaniu wydajnych instrukcji na poziomie maszynowym. Przykładowo klasa `AtomicInteger` zawiera metody `incrementAndGet` i `decrementAndGet`, które

automatycznie inkrementują i dekrementują liczby całkowite. Można na przykład bezpiecznie wygenerować szereg liczb w następujący sposób:

```
public static AtomicLong nextNumber = new AtomicLong();
// w jakimś wątku
long id = nextNumber.incrementAndGet();
```

Metoda `incrementAndGet` atomowo zwiększa wartość `AtomicLong` i zwraca tę zwiększoną wartość. Oznacza to, że nie można przeszkodzić w operacji pobrania wartości, zwiększenia jej o jeden, zapisania jej w pamięci i utworzenia nowej wartości. Mamy gwarancję, że zostanie obliczona i zwrócona poprawna wartość, nawet jeżeli obiektu będzie używać wiele wątków naraz.

Istnieją metody służące do atomowego ustawiania, dodawania i odejmowania wartości, ale jeśli trzeba wykonać jakąś bardziej skomplikowaną operację, należy się posłużyć metodą `compareAndSet`. Powiedzmy na przykład, że chcemy rejestrować największą wartość zaobserwowaną przez różne wątki. Poniższe rozwiązanie nie jest właściwe:

```
public static AtomicLong largest = new AtomicLong();
// w jakimś wątku...
largest.set(Math.max(largest.get(), observed)); // Błąd — wyścig!
```

Ta operacja nie jest niepodzielna. W zamian wartość zmiennej można zmieniać za pomocą wyrażenia lambda. W przedstawionym przykładzie moglibyśmy zastosować następujące wywołanie:

```
largest.updateAndGet(x -> Math.max(x, observed));
```

lub

```
largest.accumulateAndGet(observed, Math::max);
```

Metoda `accumulateAndGet` pobiera operator binarny, za pomocą którego dokonuje kombinacji wartości atomowej z przekazanym argumentem.

Dostępne są też metody `getAndUpdate` i `getAndAccumulate`, które zwracają starą wartość.

Definicje tych metod znajdują się też w klasach `AtomicInteger`, `AtomicIntegerArray`, `AtomicIntegerFieldUpdater`, `AtomicLongArray`, `AtomicLongFieldUpdater`, `AtomicReference`, `AtomicReferenceArray` oraz `AtomicReferenceFieldUpdater`.

Jeśli wartości atomowe są wykorzystywane przez dużą liczbę wątków, obniża się wydajność programu, ponieważ optymistyczne aktualizacje wymagają zbyt wielu prób. Problem ten rozwiązują klasy `LongAdder` i `LongAccumulator`. Klasa `LongAdder` składa się z kilku zmiennych, których suma wynosi tyle, ile bieżąca wartość. Wątki mogą zmieniać różne składniki dodawania i wraz z pojawianiem się nowych wątków automatycznie dodawane są kolejne składniki. Jest to wydajne rozwiązanie w typowym przypadku, gdy wartość sumy jest potrzebna dopiero po zakończeniu pracy. Zwiększenie wydajności często jest wyraźnie zauważalne.

Jeśli przewidywana jest ostra rywalizacja, należy się posługiwać klasą `LongAdder` zamiast `AtomicLong`. Nazwy metod w nich nieco się różnią. Metoda `increment` zwiększa licznik, `add` dodaje liczbę, a `sum` zwraca sumę.

```
var LongAdder adder = new LongAdder();
for (. . .)
  pool.submit(() -> {
    while (. . .) {
      . . .
      if (. . .) adder.increment();
    }
  });
. . .
long total = adder.sum());
```

Metoda `increment` oczywiście *nie* zwraca starej wartości. Gdyby tak robiła, niweczyłaby korzyści z dzielenia sumy na wiele składników.

Klasa `LongAccumulator` pozwala zastosować tę technikę do dowolnej operacji akumulacyjnej. W konstruktorze należy przekazać operację oraz jej neutralny element. Nowe wartości dodaje się za pomocą metody `accumulate`, a przy użyciu metody `get` można sprawdzić bieżącą wartość. Wynik działania poniższego kodu jest taki sam jak z użyciem klasy `LongAdder`:

```
var adder = new LongAccumulator(Long::sum, 0);
// w jakimś wątku...
adder.accumulate(value);
```

Wewnątrz akumulatora znajdują się zmienne a_1, a_2, ..., a_n. Każda z nich jest zainicjalizowana elementem neutralnym (w naszym przykładzie jest to `0`).

Gdy zostanie wywołana metoda `accumulate` z wartością `v`, to jedna z tych zmiennych zostanie atomowo zaktualizowana w ramach operacji $a_i = a_i \; op \; v$, gdzie *op* to operacja akumulacji zapisana w formie infiksowej. W naszym przykładzie wywołanie metody `accumulate` powoduje wykonanie obliczeń $a_i = a_i + v$ dla pewnego i.

Wynikiem operacji `get` jest $a_1 \; op \; a_2 \; op \; \ldots \; op \; a_n$. W naszym przykładzie jest to suma akumulatorów: $a_1 + a_2 + \ldots + a_n$.

Stosując inną operację, można obliczyć największą lub najmniejszą wartość. Generalnie operacja musi być łączna i przemienna. Oznacza to, że ostateczny wynik nie może zależeć od kolejności pobierania wartości pośrednich.

Istnieją też klasy `DoubleAdder` i `DoubleAccumulator`, które działają w taki sam sposób, tylko na wartościach typu `double`.

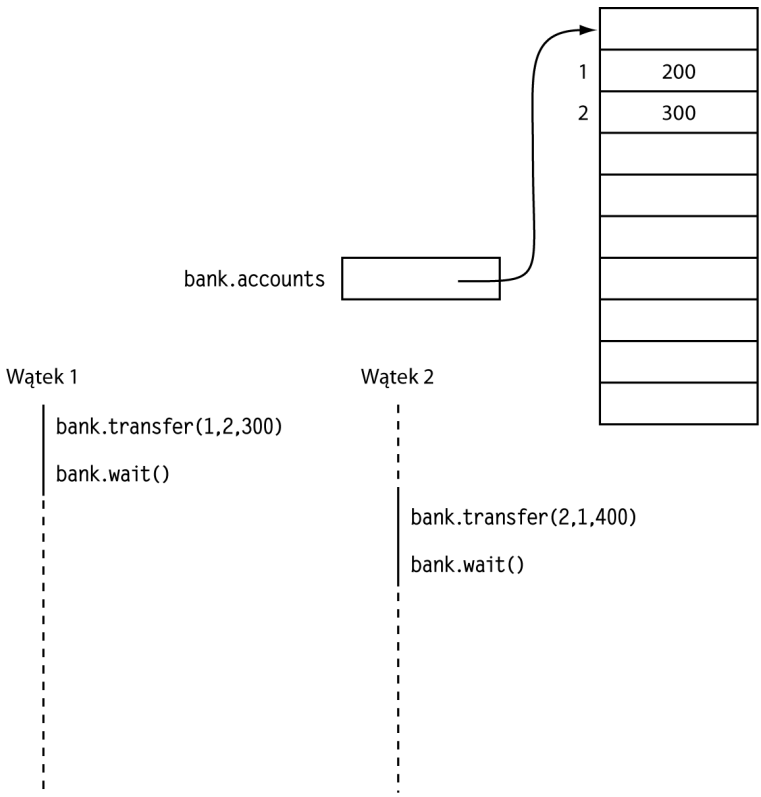
12.4.11. Zakleszczenia

Blokady i warunki nie wystarczą do rozwiązania wszystkich problemów związanych z wielowątkowością. Rozważmy następującą sytuację:

1. Konto 1: 200 dol.
2. Konto 2: 300 dol.
3. Wątek 1: przelew 300 dol. z konta 1 na konto 2
4. Wątek 2: przelew 400 dol. z konta 2 na konto 1

Jak widać na rysunku 12.4, wątki 1 i 2 są zablokowane. Żaden z nich nie może kontynuować działania, ponieważ salda na obu kontach są zbyt niskie.

Rysunek 12.4.
Zakleszczenie

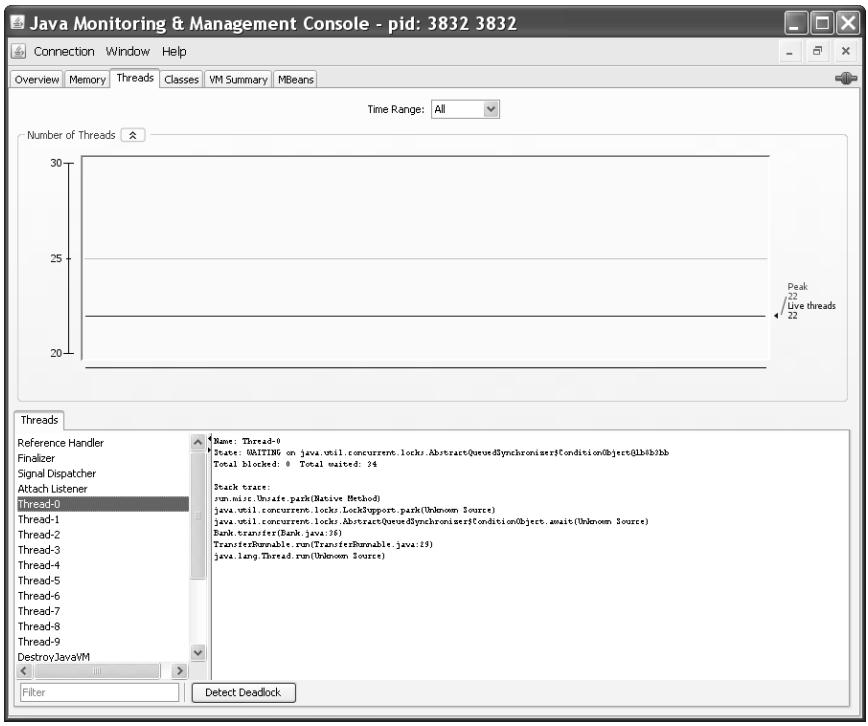


Sytuacja, w której wszystkie wątki są zablokowane, w tym przypadku (ponieważ każdy czeka na więcej pieniędzy) nazywa się **zakleszczeniem** (ang. *deadlock*).

W naszym programie zakleszczenie nie może wystąpić z prostego powodu. Kwota przelewu nie może przekraczać 1000 dolarów. Ponieważ jest sto kont, na których w sumie znajduje się 100 000 dolarów, przynajmniej jedno z nich zawsze musi zawierać więcej niż 1000 dolarów. Dzięki temu wątek przelewający pieniądze z tego konta może kontynuować działanie.

Jeśli jednak z metody `run` usuniemy ograniczenie wysokości transakcji do 1000 dolarów, zakleszczenia mogą nastąpić bardzo szybko. Można to sprawdzić. Ustaw wartość stałej `NACCOUNTS` na `10`, a parametr `max` konstruktora klasy `TransferRunnable` na `2 * INITIAL_BALANCE` i uruchom program. Podziała on przez jakiś czas i zawiesi się.

Kiedy program zawiesi się, naciśnij kombinację klawiszy *Ctrl+*, aby otrzymać listę wszystkich wątków. Każdemu wątkowi towarzyszy informacja ze stosu, dzięki czemu wiadomo, w którym miejscu jest on aktualnie zablokowany. Można też użyć narzędzia *jconsole*, które zostało opisane w rozdziale 7., i sprawdzić dane na karcie *Threads* (rysunek 12.5).



Rysunek 12.5. Karta Threads w jconsole

Innym sposobem na spowodowanie zakleszczenia jest uczynienie *i*-tego wątku odpowiedzialnym za umieszczenie pieniędzy na *i*-tym koncie zamiast za pobranie ich z *i*-tego konta. Istnieje wtedy szansa, że wszystkie wątki zbiegną się nad jednym kontem i będą próbować usunąć z niego więcej pieniędzy, niż się na nim znajduje. Aby to wypróbować, należy w programie `SynchBankTest` znaleźć metodę `run` w klasie `TransferRunnable`. W wywołaniu metody `transfer` zamień miejscami parametry `fromAccount` i `toAccount`. Uruchom program i przekonaj się, że prawie natychmiast nastąpi zakleszczenie.

Oto jeszcze jedna sytuacja, w której może łatwo dojść do zakleszczenia: w programie `Synch`
`↪BankTest` zamień metodę `signalAll` na `signal`. Po pewnym czasie program w końcu zawiesza się (tym razem także ustaw wartość `NACCOUNTS` na `10`, aby efekt był szybciej zauważalny). W przeciwieństwie do metody `signalAll`, która wysyła powiadomienie do wszystkich wątków oczekujących na zwiększenie funduszy, metoda `signal` odblokowuje tylko jeden wątek. Jeśli wątek ten nie może kontynuować, wszystkie wątki mogą zostać zablokowane. Przeanalizujmy scenariusz prezentujący, jak może dojść do zakleszczenia.

1. Konto 1: 1990 dol.
2. Pozostałe konta: po 990 dol.
3. Wątek 1: przelew 995 dol. z konta 1 na konto 2
4. Pozostałe wątki: przelew 995 dol. ze swoich kont na inne konto

Bez wątpienia wszystkie wątki poza pierwszym są zablokowane, ponieważ na ich kontach nie ma wystarczająco dużo pieniędzy.

Wątek 1 kontynuuje działanie. Dochodzimy do następującej sytuacji:

1. Konto 1: 995 dol.
2. Konto 2: 1985 dol.
3. Pozostałe konta: po 990 dol.

Wtedy wątek 1 wywołuje metodę `signal`, która odblokowuje jeden losowo wybrany wątek. Załóżmy, że padło na wątek 3. Zostaje on obudzony, stwierdza, że na jego koncie nie ma wystarczająco dużo środków, i ponownie wywołuje metodę `wait`. Jednak wątek 1 cały czas działa. Generuje nową losową transakcję, na przykład taką jak poniżej:

1. Wątek 1: przelew 997 dol. z konta 1 na konto 2

Tym razem także wątek 1 wywołuje metodę `wait` i *wszystkie* wątki są zablokowane. System ulega zakleszczeniu.

Źródłem problemów jest tutaj metoda `signal` odblokowująca tylko jeden wątek, który może nie mieć znaczenia dla utrzymania postępu programu (w naszym scenariuszu wątek 2 musi kontynuować pracę, aby pobrać pieniądze z drugiego konta).

Niestety język Java nie udostępnia żadnego mechanizmu pozwalającego uniknąć takich zakleszczeń lub je przerwać. Program musi być tak zaprojektowany, aby sytuacje tego typu nie miały szans się wydarzyć.

12.4.12. Zmienne lokalne wątków

W poprzednich podrozdziałach zostały opisane problemy, jakie można napotkać, używając w wątkach wspólnych zmiennych. Czasami można uniknąć współdzielenia, każdemu wątkowi dając egzemplarz na własność. Używa się do tego celu klasy pomocniczej `ThreadLocal`. Przykładowo klasa `SimpleDateFormat` nie jest bezpieczna wątkowo. Przypuśćmy, że mamy statyczną zmienną:

```
public static final SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
```

Jeśli dwa wątki wykonają taką operację jak poniższa:

```
String dateStamp = dateFormat.format(new Date());
```

to wynik może być niepoprawny, ponieważ wewnętrzne struktury danych używane przez `dateFormat` mogą zostać uszkodzone przez współbieżny dostęp. Jako rozwiązanie można zastosować synchronizację, która jest kosztowna, albo tworzyć lokalny obiekt `SimpleDateFormat` za każdym razem, gdy jest potrzebny. To jednak również oznacza marnotrawstwo zasobów.

Aby utworzyć po jednym egzemplarzu na wątek, należy użyć poniższego kodu:

```
public static final ThreadLocal<SimpleDateFormat> dateFormat =
  ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyy-MM-dd"));
```

Aby uzyskać dostęp do formatera, należy zastosować poniższe wywołanie:

```
String dateStamp = dateFormat.get().format(new Date());
```

Przy pierwszym wywołaniu metody `get` w wątku następuje wywołanie metody `initialValue`. Od tej pory metoda `get` zwraca egzemplarz należący do bieżącego wątku.

Podobny problem przedstawia generowanie liczb losowych w wielu wątkach. Klasa `java.util.Random` jest bezpieczna wątkowo, ale mimo to nie działa wydajnie, gdy wiele wątków musi czekać na jeden wspólny generator.

Można by było użyć klasy pomocniczej `ThreadLocal`, aby każdemu wątkowi dać osobny generator, ale w Java SE 7 udostępniono specjalną klasę, dzięki której praca ta jest wygodniejsza. Wystarczy wykonać poniższe wywołanie:

```
int random = ThreadLocalRandom.current().nextInt(upperBound);
```

Metoda `ThreadLocalRandom.current()` zwraca egzemplarz klasy `Random` dostępny tylko dla bieżącego wątku.

`java.lang.ThreadLocal<T>` **1.2**

- `T get()`

  Pobiera bieżącą wartość wątku. Jeśli metoda `get` jest wywoływana po raz pierwszy, wartość otrzymywana jest poprzez wywołanie metody `initialize`.

- `void set(T t)`

  Ustawia nową wartość wątku.

- `void remove()`

  Usuwa wartość wątku.

- `static <S> ThreadLocal<S> withInitial(Supplier<? extends S> supplier)` **8**

  Tworzy zmienną lokalną w wątku, której wartość początkowa jest utworzona przez wywołanego dostawcę.

`java.util.concurrent.ThreadLocalRandom` **7**

- `static ThreadLocalRandom current()`

  Zwraca egzemplarz klasy `Random` należący do bieżącego wątku.

12.4.13. Dlaczego metody stop i suspend są wycofywane

Początkowo w Javie dostępna była metoda `stop`, która kończyła wątek, i metoda `suspend`, która blokowała wątek do czasu, gdy inny wątek wywołał metodę `resume`. Dwie pierwsze z wymienionych metod coś łączy: obie próbują kontrolować działanie wątku, nie współpracując z nim.

Obie te metody są wycofywane. Metoda `stop` jest z gruntu niebezpieczna, a jeśli chodzi o `suspend`, to z doświadczenia wiadomo, że często prowadzi do zakleszczeń. W tym podrozdziale wyjaśniamy, dlaczego metody te sprawiają problemy i co można zrobić, aby ich uniknąć.

Zaczniemy od metody `stop`. Zamyka ona wszystkie oczekujące metody, włącznie z metodą `run`. Jeśli zostanie zastosowana na rzecz wątku, natychmiast zdejmuje on wszystkie blokady, które założył. To może prowadzić do uszkodzenia obiektów. Wyobraźmy sobie na przykład, że wątek `TransferRunnable` został zatrzymany w trakcie przelewania pieniędzy z jednego konta na inne — zdążył pobrać pieniądze, ale nie zdążył ich zapisać na drugim koncie. W tej sytuacji obiekt banku zostaje *zniszczony*. Ponieważ blokada została zdjęta, zniszczenie jest widoczne także dla innych wątków, które jeszcze nie zostały zatrzymane.

Wątek chcący zatrzymać inny wątek nie ma sposobu na sprawdzenie, kiedy wywołanie metody `stop` jest bezpieczne, a kiedy doprowadzi do zniszczenia obiektu. Dlatego odradza się używania tej metody. Aby zatrzymać wątek, należy go przerwać. Przerwany wątek może się zatrzymać wtedy, gdy jest to bezpieczne.

Niektórzy twierdzą, że metoda `stop` jest odradzana, ponieważ poprzez zatrzymywanie wątków może blokować obiekty na stałe. To jednak nieprawda. Zatrzymany wątek wychodzi z wszystkich metod synchronizowanych, które wywołał, zgłaszając wyjątek `ThreadDeath`. W rezultacie zwalniane są wszystkie wewnętrzne blokady obiektów, które wątek ten założył.

Kolej na metodę `suspend`. W przeciwieństwie do metody `stop` nie powoduje ona uszkodzenia obiektów. Jeśli jednak zawieszony zostanie wątek posiadający blokadę będzie ona niedostępna aż do odwieszenia tego wątku. Jeśli wątek, który wywołał tę metodę `suspend`, próbuje założyć tę samą blokadę, program zostaje zakleszczony — zawieszony wątek czeka na odwieszenie, a wątek, który go zawiesił, czeka na blokadę.

Sytuacje tego typu często zdarzają się w graficznych interfejsach użytkownika. Załóżmy, że mamy graficzną symulację naszego banku. Przycisk z etykietą *Wstrzymaj* zawiesza wątki dokonujące przelewów, a przycisk z etykietą *Wznów* odwiesza je.

```
pauseButton.addActionListener(event -> {
  for (int i = 0; i < threads.length; i++)
    threads[i].suspend(); // nie rób tego
});
resumeButton.addActionListener(event -> {
  for (int i = 0; i < threads.length; i++)
    threads[i].resume();
});
```

Metoda `paintComponent` będzie rysować wykres każdego konta. W tym celu utworzy tablicę sald kont za pomocą metody `getBalances`.

Jak przekonasz się w punkcie 12.7.3, zarówno akcje przycisków, jak i ponowne rysowanie odbywają się w tym samym wątku — **wątku dystrybucji zdarzeń** (ang. *event dispatch thread*). Przeanalizujmy następujący scenariusz:

1. Jeden z wątków przelewowych zakłada blokadę obiektu `bank`.
2. Użytkownik klika przycisk *Wstrzymaj*.
3. Wszystkie wątki przelewowe zostają zawieszone. Jeden z nich cały czas trzyma blokadę na obiekcie `bank`.

4. Z jakiegoś powodu konieczne jest ponowne narysowanie wykresu konta.
5. Metoda `paintComponent` wywołuje metodę `getBalances`.
6. Metoda ta próbuje założyć blokadę obiektu `bank`.

Program zostaje zamrożony.

Wątek dystrybucji zdarzeń nie może kontynuować, ponieważ blokada znajduje się w posiadaniu jednego z zawieszonych wątków. Dlatego użytkownik nie może kliknąć przycisku *Wznów* i wątki nigdy nie zostaną odwieszone.

Aby bezpiecznie zawieszać wątki, należy utworzyć zmienną `suspendRequested` i testować ją w bezpiecznym miejscu metody `run` — w takim miejscu, w którym wątek nie blokuje obiektów potrzebnych innym wątkom. Kiedy wątek odkryje, że zmienna `suspendRequested` została ustawiona, powinien czekać, aż będzie ona ponownie dostępna.

12.5. Kolekcje bezpieczne wątkowo

Jeśli kilka wątków równocześnie modyfikuje strukturę danych, na przykład tablicę skrótów, łatwo może dojść do jej zniszczenia (więcej informacji o tablicach skrótów znajduje się w rozdziale 9.). Jeden wątek może na przykład rozpocząć dodawanie elementu, ale w trakcie aktualizacji łączy między kontenerami tablicy może dojść do jego wywłaszczenia. Jeśli teraz następny wątek zacznie przeglądać tę listę, może użyć nieprawidłowych połączeń i narobić bałaganu, co może skończyć się zgłoszeniem wyjątku albo wpadnięciem w nieskończoną pętlę.

Wspólną strukturę danych można chronić za pomocą blokady, choć zwykle łatwiej jest skorzystać z implementacji bezpiecznej wątkowo. Poniżej opisujemy inne bezpieczne wątkowo kolekcje dostępne w bibliotece Javy.

12.5.1. Kolejki blokujące

Wiele problemów z wątkami można zgrabnie i bezpiecznie sformułować za pomocą jednej lub większej liczby kolejek. Wątki producenta umieszczają elementy w kolejce, a wątki konsumenta pobierają je stamtąd. Kolejka umożliwia bezpieczną wymianę danych pomiędzy wątkami. Weźmy na przykład nasz program symulujący bank. Wątki przelewające — zamiast bezpośrednio operować na obiekcie banku — wstawiają obiekty instrukcji przelewu do kolejki. Inny wątek usuwa te instrukcje i wykonuje przelewy. Tylko ten wątek ma dostęp do wnętrza obiektu banku. Nie jest potrzebna synchronizacja (oczywiście projektanci klas kolejek bezpiecznych dla wątków musieli zająć się blokadami i warunkami, ale to był ich problem, a nie nasz).

Kolejka blokująca (ang. *blocking queue*) powoduje zablokowanie wątku podczas próby dodania elementu, jeśli jest pełna, lub podczas próby usunięcia elementu, jeśli jest pusta. Kolejki tego typu znajdują zastosowanie w koordynacji działań wielu wątków. Niektóre wątki robocze mogą co jakiś czas odkładać pośrednie wyniki w kolejce blokującej, a pozostałe mogą je stamtąd usuwać i poddawać dalszej obróbce. Kolejka automatycznie kontroluje

przebieg pracy. Jeśli jeden zestaw wątków działa wolniej niż drugi, ten drugi musi poczekać na wyniki pierwszego. Jeśli pierwszy zestaw działa szybciej od drugiego, kolejka zapełnia się, dopóki drugi zestaw wątków nadąża z odbieraniem. Tabela 12.1 zawiera zestawienie metod blokujących kolejek.

Tabela 12.1. Metody kolejek blokujących

Metoda	Normalne działanie	Działanie w specjalnych warunkach
`add`	Dodaje element.	Zgłasza wyjątek `IllegalStateException`, jeśli kolejka jest pełna.
`element`	Zwraca element z czoła.	Zgłasza wyjątek `NoSuchElementException`, jeśli kolejka jest pusta.
`offer`	Dodaje element i zwraca wartość `true`.	Zwraca wartość `false`, jeśli kolejka jest pełna.
`peek`	Zwraca element z czoła.	Zwraca wartość `null`, jeśli kolejka jest pusta.
`poll`	Usuwa i zwraca element z czoła.	Zwraca wartość `null`, jeśli kolejka jest pusta.
`put`	Dodaje element.	Blokuje, jeśli kolejka jest pełna.
`remove`	Usuwa i zwraca element z czoła.	Zgłasza wyjątek `NoSuchElementException`, jeśli kolejka jest pusta.
`take`	Usuwa i zwraca element z czoła.	Blokuje, jeśli kolejka jest pusta.

Metody kolejek blokujących można podzielić na trzy kategorie w zależności od działania, kiedy kolejka jest pełna lub pusta. Jeśli kolejka jest wykorzystywana jako narzędzie do zarządzania wątkami, należy używać metod `put` i `take`. Metody `add`, `remove` i `element` zgłaszają wyjątek, kiedy element jest dodawany do pełnej kolejki lub pobierany z pustej. Oczywiście w programie wielowątkowym kolejka może się zapełnić i zrobić pusta w każdej chwili. Dlatego w takich sytuacjach należy używać metod `offer`, `poll` i `peek`. Metody te nie zgłaszają wyjątku, tylko zwracają wartość oznaczającą niepowodzenie operacji, jeśli zakończą się niepowodzeniem.

Metody `poll` i `peek` informują o niepowodzeniu za pomocą wartości zwrotnej `null`. Dlatego do tego typu kolejek nie można wstawiać referencji `null`.

Istnieją także wersje czasowe metod `offer` i `poll`. Na przykład poniższa instrukcja:

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

przez sto milisekund próbuje wstawić element do ogona kolejki. Jeśli się jej powiedzie, zwróci wartość `true`, w przeciwnym przypadku, jeśli nie wykona operacji w wyznaczonym czasie, zwróci `false`. Podobnie instrukcja:

```
Object head = q.poll(100, TimeUnit.MILLISECONDS)
```

przez sto milisekund próbuje usunąć element z czoła kolejki. Jeśli się jej powiedzie, zwróci ten element, w przeciwnym przypadku, jeśli nie wykona operacji w wyznaczonym czasie, zwróci `false`.

Metoda `put` włącza blokadę, jeśli kolejka jest pełna, a metoda `take` robi to samo, gdy kolejka jest pusta. Metody te są odpowiednikami metod `offer` i `poll` bez ograniczenia czasowego.

W pakiecie `java.util.concurrent` znajduje się kilka wersji kolejek blokujących. Kolejka `LinkedBlockingQueue` nie posiada domyślnej górnej granicy pojemności, ale można ją określić. Jej dwustronna wersja to `LinkedBlockingDeque`. Kolejka `ArrayBlockingQueue` ma określoną pojemność i opcjonalny parametr włączający wymóg uczciwości. Jeśli kolejka jest uczciwa, preferencyjnie traktowane są te wątki, które czekają najdłużej. Należy jednak pamiętać, że uczciwość zawsze powoduje straty szybkości, przez co opcję tę powinno się stosować wyłącznie wtedy, gdy jest to całkowicie niezbędne.

Kolejka `PriorityBlockingQueue` jest kolejką priorytetową, nie typu „pierwszy wszedł, pierwszy wyszedł". Elementy są usuwane zgodnie z ich priorytetami. Kolejka ta ma nieograniczoną pojemność, ale pobieranie elementów z pustej konstrukcji powoduje blokadę (więcej informacji na temat kolejek priorytetowych znajduje się w rozdziale 9.).

W końcu kolejka `DelayQueue` przechowuje obiekty, które implementują interfejs `Delayed`:

```
interface Delayed extends Comparable<Delayed>
{
  long getDelay(TimeUnit unit);
}
```

Metoda `getDelay` zwraca ilość pozostałego czasu opóźnienia obiektu. Wartość ujemna oznacza, że czas ten upłynął. Elementy z tej kolejki mogą zostać usunięte dopiero wtedy, gdy upłynie określony czas opóźnienia. Konieczna jest także implementacja metody `compareTo`. Kolejka `DelayQueue` używa tej metody do sortowania elementów.

W Java SE 7 dodano interfejs `TransferQueue` pozwalający wątkowi producenta poczekać, aż konsument będzie gotowy do przyjęcia elementu. Gdy producent wywołuje poniższą metodę:

```
q.transfer(item);
```

wywołanie to zostaje zablokowane do czasu, aż blokadę usunie inny wątek. Opisywany interfejs jest zaimplementowany w klasie `LinkedTransferQueue`.

Program przedstawiony na listingu 12.6 demonstruje sposób kontroli zestawu wątków za pomocą kolejki blokującej. Przeszukuje on wszystkie pliki znajdujące się w katalogu i jego podkatalogach oraz drukuje linijki, które zawierają dane słowo kluczowe.

Listing 12.6. blockingQueue/BlockingQueueTest.java

```
package blockingQueue;

import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;

/**
 * @version 1.03 2018-03-17
 * @author Cay Horstmann
 */
public class BlockingQueueTest
{
   private static final int FILE_QUEUE_SIZE = 10;
```

```
private static final int SEARCH_THREADS = 100;
private static final Path DUMMY = Path.of("");
private static BlockingQueue<Path> queue = new Array
↪BlockingQueue<>(FILE_QUEUE_SIZE);

public static void main(String[] args)
{
   try (var in = new Scanner(System.in))
   {
      System.out.print("Enter base directory (e.g. /opt/jdk-9-src): ");
      String directory = in.nextLine();
      System.out.print("Wpisz słowo kluczowe (np. volatile): ");
      String keyword = in.nextLine();

      Runnable enumerator = () -> {
         try
         {
            enumerate(Path.of(directory));
            queue.put(DUMMY);
         }
         catch (IOException e)
         {
            e.printStackTrace();
         }
         catch (InterruptedException e)
         {
         }
      };

      new Thread(enumerator).start();
      for (int i = 1; i <= SEARCH_THREADS; i++) {
         Runnable searcher = () -> {
            try
            {
               var done = false;
               while (!done)
               {
                  Path file = queue.take();
                  if (file == DUMMY)
                  {
                     queue.put(file);
                     done = true;
                  }
                  else search(file, keyword);
               }
            }
            catch (IOException e)
            {
               e.printStackTrace();
            }
            catch (InterruptedException e)
            {
            }
         };
         new Thread(searcher).start();
      }
   }
```

```
   }

   /**
    * Rekurencyjnie przegląda wszystkie pliki w danym katalogu i jego podkatalogach.
    * Opis operacji na strumieniach i plikach znajduje się w rozdziałach 1. i 2. tomu II.
    * @param directory katalog początkowy
    */
   public static void enumerate(Path directory) throws IOException,
   ↪InterruptedException
   {
      try (Stream<Path> children = Files.list(directory))
      {
         for (Path child : children.collect(Collectors.toList()))
         {
            if (Files.isDirectory(child))
               enumerate(child);
            else
               queue.put(child);
         }
      }
   }

   /**
    * Szuka słowa kluczowego w pliku i drukuje wszystkie zawierające je wiersze tekstu.
    * @param file plik do przeszukania
    * @param keyword słowo do znalezienia
    */
   public static void search(Path file, String keyword) throws IOException
   {
      try (var in = new Scanner(file, StandardCharsets.UTF_8))
      {
         int lineNumber = 0;
         while (in.hasNextLine())
         {
            lineNumber++;
            String line = in.nextLine();
            if (line.contains(keyword))
               System.out.printf("%s:%d:%s%n", file, lineNumber, line);
         }
      }
   }
}
```

Wątek producenta (producent) tworzy wyliczenie wszystkich plików znalezionych we wszystkich podkatalogach i wstawia je do kolejki blokującej. Operacja ta jest bardzo szybka i gdyby nie ograniczenie pojemności, kolejka w szybkim tempie zapełniłaby się wszystkimi plikami znajdującymi się w systemie plików.

Uruchamiamy także dużą liczbę wątków przeszukujących. Każdy taki wątek pobiera plik z kolejki, otwiera go, drukuje wszystkie linijki zawierające dane słowo kluczowe i pobiera następny plik. Do zakończenia aplikacji, kiedy dalsza jej praca jest już zbędna, wykorzystaliśmy pewną sztuczkę. Wątek wyliczeniowy sygnalizuje ukończenie pracy, umieszczając w kolejce atrapę obiektu (przypomina to umieszczanie walizki z etykietą „Ostatnia torba” na końcu taśmy z walizkami na lotnisku). Kiedy wątek przeszukujący pobierze taki obiekt, odkłada go z powrotem i kończy działanie.

Zwróć uwagę, że nie trzeba bezpośrednio stosować synchronizacji. W tej aplikacji do synchronizacji używamy kolejki.

`java.util.concurrent.ArrayBlockingQueue<E>` **5.0**

- `ArrayBlockingQueue(int capacity)`
- `ArrayBlockingQueue(int capacity, boolean fair)`

  Tworzy kolejkę blokującą o określonej pojemności i z ustawioną zasadą uczciwości. Kolejka ta jest zaimplementowana jako tablica cykliczna.

`java.util.concurrent.LinkedBlockingQueue<E>` **5.0**
`java.util.concurrent.LinkedBlockingDeque<E>` **6**

- `LinkedBlockingQueue()`
- `LinkedBlockingDeque()`

  Tworzy nieograniczoną kolejkę blokującą jedno- lub dwustronną, zaimplementowaną jako lista powiązana.

- `LinkedBlockingQueue(int capacity)`
- `LinkedBlockingDeque(int capacity)`

  Tworzy ograniczoną kolejkę jedno- lub dwustronną blokującą o określonej pojemności, zaimplementowaną jako lista powiązana.

`java.util.concurrent.DelayQueue<E extends Delayed>` **5.0**

- `DelayQueue()`

  Tworzy nieograniczoną kolejkę elementów typu `Delayed`. Z kolejki tej można usuwać tylko te elementy, których czas opóźnienia upłynął.

`java.util.concurrent.Delayed` **5.0**

- `long getDelay(TimeUnit unit)`

  Zwraca opóźnienie obiektu mierzone w określonej jednostce czasu.

`java.util.concurrent.PriorityBlockingQueue<E>` **5.0**

- `PriorityBlockingQueue()`
- `PriorityBlockingQueue(int initialCapacity)`
- `PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)`

  Tworzy nieograniczoną priorytetową kolejkę blokującą zaimplementowaną jako sterta. Domyślna pojemność początkowa (`initialCapacity`) wynosi `11`. Jeśli komparator nie zostanie podany, elementy muszą implementować interfejs `Comparable`.

`java.util.concurrent.BlockingQueue<E>` **5.0**

- `void put(E element)`

  Dodaje element i w razie konieczności włącza blokowanie.

- `E take()`

  Usuwa i zwraca element z czoła, w razie konieczności włącza blokowanie.

- `boolean offer(E element, long time, TimeUnit unit)`

  Dodaje określony element i zwraca wartość `true`, jeśli operacja zakończy się powodzeniem. W razie konieczności włącza blokowanie, aż element zostanie dodany lub upłynie określony czas.

- `E poll(long time, TimeUnit unit)`

  Usuwa i zwraca element z czoła. W razie konieczności włącza blokowanie, aż element będzie dostępny lub upłynie określony czas. W razie niepowodzenia zwraca wartość `null`.

`java.util.concurrent.BlockingDeque<E>` **6**

- `void putFirst(E element)`
- `void putLast(E element)`

  Dodaje element i w razie potrzeby włącza blokadę.

- `E takeFirst()`
- `E takeLast()`

  Usuwa i zwraca element z czoła lub ogona i w razie potrzeby włącza blokadę.

- `boolean offerFirst(E element, long time, TimeUnit unit)`
- `boolean offerLast(E element, long time, TimeUnit unit)`

  Dodaje określony element i zwraca wartość `true`, jeśli operacja zakończy się powodzeniem. W razie potrzeby włącza blokadę, aż element zostanie dodany albo upłynie wyznaczony czas.

- `E pollFirst(long time, TimeUnit unit)`
- `E pollLast(long time, TimeUnit unit)`

  Usuwa i zwraca element z czoła lub ogona. W razie potrzeby włącza blokadę, aż element będzie dostępny lub upłynie wyznaczony czas. W przypadku niepowodzenia zwraca wartość `null`.

`java.util.concurrent.TransferQueue<E>` **7**

- `void transfer(E element)`
- `boolean tryTransfer(E element, long time, TimeUnit unit)`

  Przesyła wartość albo próbuje ją przesłać w określonym czasie, zakładając blokadę do czasu, aż inny wątek usunie element. Druga metoda zwraca `true` w razie powodzenia.

12.5.2. Szybkie słowniki, zbiory i kolejki

W pakiecie `java.util.concurrent` znajdują się następujące szybkie implementacje słowników, zbiorów uporządkowanych i kolejek: `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet` oraz `ConcurrentLinkedQueue`.

W kolekcjach tych zastosowano zaawansowane algorytmy minimalizujące rywalizację wątków poprzez umożliwianie równoległego dostępu do różnych części struktury danych.

W przeciwieństwie do większości kolekcji, w tych metoda `size` niekoniecznie działa w stałym czasie. Określenie aktualnego rozmiaru tych kolekcji zazwyczaj wymaga ich przemierzenia.

W niektórych aplikacjach wykorzystywane są tak ogromne współbieżne słowniki skrótów, że metoda `size` sobie z nimi nie radzi, ponieważ zwraca wartości typu `int`. Co może zrobić ktoś, kto ma słownik zawierający ponad dwa miliardy elementów? Metoda `mappingCount` zwraca rozmiar w postaci liczby typu `long`.

Kolekcje te zwracają tak zwane **słabo spójne iteratory** (ang. *weakly consistent iterators*). Oznacza to, że mogą one (choć nie muszą) odzwierciedlać wszystkie modyfikacje dokonane po ich skonstruowaniu. Nie zwracają one jednak dwukrotnie wartości i nie zgłaszają wyjątku `ConcurrentModificationException`.

W przeciwieństwie do opisywanych iteratorów, iteratory kolekcji z pakietu `java.util` zgłaszają wyjątek `ConcurrentModificationException`, jeśli kolekcja zostanie zmodyfikowana po ich utworzeniu.

Struktura `ConcurrentHashMap` jest zdolna szybko obsłużyć dużą liczbę czytników i ustaloną liczbę algorytmów zapisujących. Domyślnie założono, że może być do 16 algorytmów zapisujących *działających jednocześnie*. Może być ich więcej, ale jeśli więcej niż 16 z nich zapisuje w tym samym czasie, reszta pozostaje tymczasowo zablokowana. Można podać większą liczbę w konstruktorze, ale istnieje niewielkie prawdopodobieństwo, że będzie to potrzebne.

Słownik skrótów przechowuje wszystkie elementy o takiej samej wartości skrótu w jednym „kubełku". W niektórych aplikacjach wykorzystywane są niskiej jakości funkcje mieszające, w wyniku czego wszystkie elementy trafiają do niewielkiej liczby kubełków, co radykalnie pogarsza wydajność programu. Nawet w miarę niezłe funkcje mieszające, np. używana w klasie `String`, mogą stwarzać problemy. Haker może na przykład spowolnić działanie programu przez spreparowanie dużej liczby łańcuchów o takiej samej wartości skrótu. W nowych wersjach Javy współbieżny słownik skrótów organizuje kubełki jako drzewa, a nie listy, kiedy typ klucza implementuje interfejs `Comparable`, co zapewnia gwarantowaną wydajność $O(\log(n))$.

`java.util.concurrent.ConcurrentLinkedQueue<E>` **5.0**

- `ConcurrentLinkedQueue<E>()`

  Tworzy nieograniczoną kolejkę nieblokującą, którą można bezpiecznie przetwarzać w wielu wątkach.

`java.util.concurrent.ConcurrentSkipListSet<E>` **6**

- `ConcurrentSkipListSet<E>()`
- `ConcurrentSkipListSet<E>(Comparator<? super E> comp)`

  Tworzy zbiór uporządkowany, do którego można bezpiecznie uzyskać dostęp w wielu wątkach. Pierwszy z konstruktorów wymaga, aby elementy implementowały interfejs `Comparable`.

`java.util.concurrent.ConcurrentHashMap<K, V>` **5.0**
`java.util.concurrent.ConcurrentSkipListMap<K, V>` **6**

- `ConcurrentHashMap<K, V>()`
- `ConcurrentHashMap<K, V>(int initialCapacity)`
- `ConcurrentHashMap<K, V>(int initialCapacity, float loadFactor, int concurrencyLevel)`

  Tworzy słownik skrótów, do którego można bezpiecznie uzyskać dostęp w wielu wątkach. Domyślna pojemność początkowa (`initialCapacity`) wynosi `16`. Jeśli średnie zapełnienie komórki przekracza ten współczynnik zapełnienia, rozmiar tablicy jest zmieniany. Domyślna wartość to 0,75. Parametr `concurrencyLevel` określa przewidywaną liczbę współbieżnych wątków zapisujących.

- `ConcurrentSkipListMap<K, V>()`
- `ConcurrentSkipListSet<K, V>(Comparator<? super K> comp)`

  Tworzy uporządkowany słownik, do którego można uzyskać bezpieczny dostęp w wielu wątkach. Pierwszy z konstruktorów wymaga, aby klucze implementowały interfejs `Comparable`.

12.5.3. Atomowe modyfikowanie elementów słowników

Oryginalna wersja słownika `ConcurrentHashMap` zawierała tylko kilka metod do atomowego wykonywania modyfikacji, przez co programiści musieli stosować różne nieeleganckie rozwiązania. Powiedzmy, że chcemy policzyć częstość występowania czegoś, np. kilka wątków napotyka różne słowa i chcemy się dowiedzieć, jak często każde z nich występuje.

Czy możemy wykorzystać strukturę `ConcurrentHashMap<String, Long>`? Spójrz na poniższy kod, w którym zwiększany jest licznik. Oczywiście nie jest to rozwiązanie bezpieczne pod kątem wątków:

```
Long oldValue = map.get(word);
Long newValue = oldValue == null ? 1 : oldValue + 1;
map.put(word, newValue); // Błąd — może nie zamienić oldValue.
```

W tym samym czasie inny wątek może zmieniać wartość tego samego licznika.

Niektórzy programiści dziwią się, że rzekomo bezpieczna pod kątem wątków struktura danych umożliwia wykonywanie operacji, które nie są bezpieczne wątkowo. Trzeba jednak pamiętać, że są różne punkty widzenia. Jeśli kilka wątków modyfikuje zwykły słownik `HashMap`, to mogą zniszczyć wewnętrzną strukturę (tablicę list powiązanych). Niektóre z łączy mogą się zagubić albo zapętlić, przez co struktura stanie się bezużyteczna. Takie coś nigdy nie zdarzy się w przypadku słownika `ConcurrentHashMap`. W powyższym przykładzie metody `get` i `put` na pewno nie uszkodzą samej struktury danych. Jednak sekwencja wykonywanych operacji nie jest niepodzielna, więc wynik jest nieprzewidywalny.

W starszych wersjach Javy stosowano sztuczkę z użyciem metody `replace`, która atomowo zamieniała starą wartość na nową, pod warunkiem że żaden inny wątek nie zrobił tego wcześniej. Próby należało powtarzać, aż się uda:

```
do
{
  oldValue = map.get(word);
  newValue = oldValue == null ? 1 : oldValue + 1;
} while (!map.replace(word, oldValue, newValue));
```

Ewentualnie można użyć słownika `ConcurrentHashMap<String, AtomicLong>`. Wówczas należałoby napisać taką procedurę:

```
map.putIfAbsent(word, new AtomicLong());
map.get(word).incrementAndGet();
```

Niestety w każdej inkrementacji tworzony jest nowy obiekt `AtomicLong`, niezależnie od tego, czy jest potrzebny, czy nie.

Obecnie API Javy zawiera metody, dzięki którym można wykonywać atomowe aktualizacje w prostszy sposób. Metodę `compute` wywołuje się z kluczem i funkcją mającą obliczyć nową wartość. Funkcja ta otrzymuje ów klucz i związaną z nim wartość albo `null`, jeśli taki klucz nie istnieje, i oblicza nową wartość. Poniżej znajduje się przykład aktualizacji słownika liczników całkowitoliczbowych:

```
map.compute(word, (k, v) -> v == null ? 1 : v + 1);
```

W strukturze `ConcurrentHashMap` nie można przechowywać wartości `null`. Warto o tym pamiętać, ponieważ wiele metod wykorzystuje tę wartość jako wskazówkę, że danego klucza nie ma w słowniku.

Istnieją też wersje metod `computeIfPresent` i `computeIfAbsent`, które obliczają nową wartość tylko wtedy, gdy istnieje już stara lub gdy stara nie istnieje. Słownik liczników typu `LongAdder` można zaktualizować w następujący sposób:

```
map.computeIfAbsent(word, k -> new LongAdder()).increment();
```

To wywołanie jest prawie identyczne z wcześniej pokazywanym wywołaniem metody `putIf`
`↪Absent`, ale konstruktor `LongAdder` zostaje wywołany tylko wtedy, gdy rzeczywiście jest potrzebny nowy licznik.

Przy pierwszym dodawaniu klucza często konieczne jest wykonanie jakiejś specjalnej czynności. Szczególnie przydatna w takiej sytuacji jest metoda `merge`, posiadająca parametr umożliwiający przekazanie wartości początkowej, która zostanie użyta, gdy klucza jeszcze nie będzie. W przeciwnym przypadku zostanie wywołana przekazana przez programistę funkcja mająca za zadanie utworzyć kombinację z wartości istniejącej i początkowej. (W odróżnieniu od metody `compute`, funkcja *nie* przetwarza klucza).

```
map.merge(word, 1L, (existingValue, newValue) -> existingValue + newValue);
```

Można też napisać to prościej:

```
map.merge(word, 1L, Long::sum);
```

Zwięźlej już się nie da.

Jeśli funkcja przekazana do metody `compute` lub `merge` zwraca `null`, istniejący wpis zostaje usunięty ze słownika.

Pamiętaj, że funkcje przekazywane do metod `compute` i `merge` nie powinny być zbyt rozbudowane, ponieważ w czasie ich działania mogą być zablokowane inne operacje na słowniku. Oczywiście funkcje te nie powinny też zmieniać niczego w innych częściach słownika.

Program pokazany na listingu 12.7 liczy słowa w plikach Javy znajdujących się w drzewie katalogów przy użyciu współbieżnego słownika skrótów.

Listing 12.7. concurrentHashMap/CHMDemo.java

```
package concurrentHashMap;

import java.io.*;
import java.nio.file.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;

/**
 * Program demonstrujący użycie słowników skrótów.
 * @version 1.0 2018-01-04
 * @author Cay Horstmann
 */
public class CHMDemo
{
   public static ConcurrentHashMap<String, Long> map = new ConcurrentHashMap<>();

   /**
    * Dodaje wszystkie słowa z pliku do współbieżnego słownika skrótów.
    * @param file plik
    */
   public static void process(Path file)
   {
      try (var in = new Scanner(file))
      {
```

```
            while (in.hasNext())
            {
               String word = in.next();
               map.merge(word, 1L, Long::sum);
            }
         }
         catch (IOException e)
         {
            e.printStackTrace();
         }
      }

      /**
       * Zwraca wszystkie podkatalogi danego katalogu — zobacz rozdziały 1. i 2. w tomie II.
       * @param rootDir katalog główny
       * @return zbiór wszystkich podkatalogów katalogu głównego
       */
      public static Set<Path> descendants(Path rootDir) throws IOException
      {
         try (Stream<Path> entries = Files.walk(rootDir))
         {
            return entries.collect(Collectors.toSet());
         }
      }

      public static void main(String[] args)
            throws InterruptedException, ExecutionException, IOException
      {
         int processors = Runtime.getRuntime().availableProcessors();
         ExecutorService executor = Executors.newFixedThreadPool(processors);
         Path pathToRoot = Path.of(".");
         for (Path p : descendants(pathToRoot))
         {
            if (p.getFileName().toString().endsWith(".java"))
               executor.execute(() -> process(p));
         }
         executor.shutdown();
         executor.awaitTermination(10, TimeUnit.MINUTES);
         map.forEach((k, v) ->
            {
               if (v >= 10)
                  System.out.println(k + " occurs " + v + " times");
            });
      }
   }
```

12.5.4. Operacje masowe na współbieżnych słownikach skrótów

W API Javy zapewniono możliwość wykonywania na współbieżnych słownikach skrótów operacji masowych, które są bezpieczne nawet wówczas, gdy na słowniku operują inne wątki. Operacja masowa przegląda słownik i wykonuje działania na elementach, które znajdzie. Nie trzeba tworzyć żadnego stałego obrazu słownika. Jeśli nie ma pewności, czy słownik nie jest modyfikowany w czasie działania operacji masowej, jej wynik należy traktować jako przybliżenie stanu słownika.

Istnieją trzy rodzaje operacji:

- `search` wywołuje funkcję na każdym kluczu i/lub każdej wartości, aż funkcja ta zwróci wynik różny od `null`. Następnie szukanie kończy się i następuje zwrócenie wyniku funkcji.
- `reduce` tworzy kombinację wszystkich kluczy i/lub wartości za pomocą przekazanej funkcji akumulacyjnej.
- `forEach` stosuje funkcję do wszystkich kluczy i/lub wartości.

Każda operacja występuje w czterech wersjach:

- *operacja*`Keys` — działa na kluczach;
- *operacja*`Values` — działa na wartościach;
- *operacja* — działa na kluczach i wartościach;
- *operacja*`Entries` — działa na obiektach typu `Map.Entry`.

Z każdą z tych operacji trzeba określić **próg zrównoleglania**. Jeśli słownik zawiera więcej elementów niż określony limit, następuje zrównoleglenie operacji masowej. Jeśli operacja ma być wykonywana w jednym wątku, należy zastosować próg `Long.MAX_VALUE`. Jeżeli chcesz, aby operacja dysponowała maksymalną liczbą wątków, zdefiniuj próg `1`.

Najpierw przyjrzymy się rodzinie metod `search`:

```
U searchKeys(long threshold, BiFunction<? super K, ? extends U> f)
U searchValues(long threshold, BiFunction<? super V, ? extends U> f)
U search(long threshold, BiFunction<? super K, ? super V,? extends U> f)
U searchEntries(long threshold, BiFunction<Map.Entry<K, V>, ? extends U> f)
```

Powiedzmy na przykład, że chcemy znaleźć pierwsze słowo, które występuje w tekście więcej niż 1000 razy. W tym celu musimy przeszukać klucze i wartości:

```
String result = map.search(threshold, (k, v) -> v > 1000 ? k : null);
```

Zmienna `result` zostanie ustawiona na pierwsze znalezione słowo lub `null`, jeśli funkcja `search` zwróci `null` dla wszystkich danych wejściowych.

Metoda `forEach` występuje w dwóch wariantach. Pierwszy po prostu stosuje funkcję konsumującą do każdego elementu słownika, np.:

```
map.forEach(threshold,
  (k, v) -> System.out.println(k + " -> " + v));
```

Drugi natomiast przyjmuje dodatkowo funkcję *przekształcającą*, która zostaje zastosowana jako pierwsza, aby jej wynik został przekazany do funkcji konsumującej:

```
map.forEach(threshold,
  (k, v) -> k + " -> " + v, // przekształcenie
  System.out::println); // konsumpcja
```

Funkcja przekształcająca może pełnić rolę filtra. Jeśli zwróci `null`, wartość ta jest po cichu pomijana. Poniżej na przykład drukujemy tylko wpisy zawierające duże wartości:

```
map.forEach(threshold,
(k, v) -> v > 1000 ? k + " -> " + v : null, // filtr i przekształcenie
System.out::println); // wartości null nie są przekazywane do konsumenta
```

Operacje `reduce` tworzą kombinacje otrzymywanych danych za pomocą funkcji akumulacyjnej. Poniżej na przykład obliczamy sumę wszystkich wartości:

```
Long sum = map.reduceValues(threshold, Long::sum);
```

Tak jak w przypadku `forEach`, można też dodatkowo przekazać funkcję przekształcającą. Poniżej obliczamy długość najdłuższego klucza:

```
Integer maxlength = map.reduceKeys(threshold,
  String::length, // przekształcenie
  Integer::max);  // akumulacja
```

Funkcja przekształcająca pełni rolę filtra, który zwraca `null` w celu wykluczenia niechcianych wartości. W poniższym przykładzie obliczamy, ile elementów ma wartość większą niż `1000`:

```
Long count = map.reduceValues(threshold,
  v -> v > 1000 ? 1L : null,
  Long::sum);
```

Jeżeli słownik jest pusty lub wszystkie elementy zostaną odfiltrowane, operacja `reduce` zwraca wartość `null`. Jeśli jest tylko jeden element, zostaje zwrócone jego przekształcenie i funkcja akumulacyjna nie jest wywoływana.

Istnieją specjalizacje zwracające wartości typów `int`, `long` i `double` z przyrostkami `ToInt`, `ToLong` i `ToDouble`. Aby móc z nich skorzystać, należy przekształcić dane wejściowe na typ podstawowy oraz określić wartość domyślną i funkcję akumulacyjną. Wartość domyślna jest zwracana, gdy słownik jest pusty.

```
long sum = map.reduceValuesToLong(threshold,
Long::longValue, // przekształcenie na typ podstawowy
0, // domyślna wartość dla pustego słownika
Long::sum); // akumulator wartości typu podstawowego
```

Specjalizacje te działają inaczej od wersji obiektowych, gdy w strukturze jest tylko jeden element. Nie zwracają przekształconego elementu, lecz tworzą jego kombinację z wartością domyślną. Dlatego też wartość ta musi być neutralna.

12.5.5. Współbieżne widoki zbiorów

Powiedzmy, że zamiast słownika chcemy utworzyć duży i bezpieczny pod względem wątków zbiór. W Javie nie ma klasy `ConcurrentHashSet` i wiadomo, że próba szybkiego jej napisania nie jest dobrym pomysłem. Oczywiście zawsze można użyć słownika `Concurrent↪HashMap` ze zmyślonymi wartościami, ale wówczas otrzymamy słownik, a nie zbiór, przez co nie będziemy mogli posługiwać się metodami z interfejsu `Set`.

Statyczna metoda newKeySet zwraca obiekt typu Set<K>, będący opakowaniem dla ConcurrentHashMap<K, Boolean>. (Wszystkie wartości słownika to Boolean.TRUE, ale to nas nie obchodzi, ponieważ i tak używamy go jako zbioru).

```
Set<String> words = ConcurrentHashMap.<String>newKeySet();
```

Oczywiście jeśli mamy już słownik, metoda keySet zwraca zbiór kluczy. Zbiór ten można modyfikować. Gdy usuwa się z niego elementy, to ze słownika usuwane są klucze (i ich wartości). Nie ma jednak sensu dodawać elementów do zbioru kluczy, ponieważ nie byłoby odpowiadających im wartości do dodania. W klasie ConcurrentHashMap zdefiniowano jeszcze jedną metodę keySet z wartością domyślną, która jest używana przy dodawaniu elementów do zbioru:

```
Set<String> words = map.keySet(1L);
words.add("Java");
```

Jeśli do tej pory w zbiorze words nie było klucza "Java", to teraz ma on przypisaną wartość 1.

12.5.6. Tablice kopiowane przy zapisie

CopyOnWriteArrayList i CopyOnWriteArraySet to bezpieczne wątkowo kolekcje, których mutatory tworzą kopie tablic. Taki sposób działania sprawdza się w sytuacjach, w których liczba wątków iterujących po kolekcji znacznie przewyższa liczbę wątków ją modyfikujących. Utworzony iterator zawiera referencję do aktualnej tablicy. Jeśli tablica ta zostanie później zmodyfikowana, iterator ten nadal będzie miał starą tablicę, mimo że tablica kolekcji jest zamieniona. Dzięki temu starszy iterator dysponuje spójnym (choć potencjalnie przestarzałym) widokiem, do którego ma dostęp nieobciążony żadnym dodatkowym narzutem synchronizacji.

12.5.7. Równoległe algorytmy tablicowe

Klasa Arrays dodano kilka operacji równoległych. Jedną z nich jest statyczna metoda Arrays.
➥parallelSort, która sortuje tablicę wartości typów podstawowych lub obiektów. Na przykład:

```
String contents = new String(Files.readAllBytes(
Path.of("alice.txt")), StandardCharsets.UTF_8); // wczytanie pliku do łańcucha
String[] words = contents.split("[\\P{L}]+"); // dzielenie wg znaków innych niż litery
Arrays.parallelSort(words);
```

Do sortowania obiektów można użyć komparatora.

```
Arrays.parallelSort(words, Comparator.comparing(String::length));
```

We wszystkich metodach jest możliwość określenia granic zakresu, np.:

```
values.parallelSort(values.length / 2, values.length); // sortuje górną połowę
```

Początkowo słowo parallel (równoległy) w nazwach tych metod może dziwić, ponieważ użytkownika nie powinno interesować, w jaki sposób wykonywane jest sortowanie. Jednak projektanci API chcieli, aby było jasne, że chodzi o sortowanie równoległe. Dzięki temu użytkownik metody od razu wie, że nie powinien stosować komparatora z efektami ubocznymi.

Metoda `parallelSetAll` wstawia do tablicy wartości obliczone przez funkcję. Funkcja ta pobiera indeks elementu i oblicza wartość dla tej lokalizacji.

```
Arrays.parallelSetAll(values, i -> i % 10);
  // wstawia do values wartości 0 1 2 3 4 5 6 7 8 9 0 1 2 . . .
```

Bez wątpienia w przypadku tej operacji zrównoleglenie jest korzystne. Istnieją wersje tej metody dla wszystkich tablic typów podstawowych i obiektowych.

Istnieje też metoda o nazwie `parallelPrefix`, która zamienia każdy element tablicy na jego kombinację z poprzednim elementem za pomocą podanej operacji. Że co? Najlepiej obejrzeć przykład. Mamy tablicę [1, 2, 3, 4, …] i operację ×. Po wykonaniu operacji `Arrays.parallelPrefix(values, (x, y) -> x * y)` zawartość tej tablicy wygląda następująco:

```
[1, 1
× 2, 1
× 2
× 3, 1
× 2
× 3
× 4, …]
```

Pewnie się zdziwisz, ale takie obliczenia można poddać zrównolegleniu. Najpierw łączymy sąsiednie elementy, jak poniżej:

```
[1, 1
× 2, 3, 3
× 4, 5, 5
× 6, 7, 7
× 8]
```

Jaśniejsze wartości pozostają nienaruszone. Niewątpliwie obliczenia te można wykonać równolegle w osobnych obszarach tablicy. W następnym kroku aktualizujemy wskazane elementy, mnożąc je przez elementy znajdujące się o jedną lub dwie pozycje niżej:

```
[1, 1 × 2
, 1 × 2 × 3, 1 × 2 × 3 × 4, 5, 5 × 6, 5 × 6 × 7, 5 × 6 × 7 × 8]
```

To także można zrobić równolegle. Po log *n* krokach proces jest zakończony. Jeśli komputer dysponuje wystarczającą liczbą procesorów, taki sposób wykonywania obliczeń jest efektywniejszy niż zwykłe liniowe liczenie. W specjalistycznym sprzęcie taki algorytm jest wykorzystywany bardzo często, a jego użytkownicy znajdują dla niego bardzo ciekawe zastosowania.

12.5.8. Starsze kolekcje bezpieczne wątkowo

Od samego początku istnienia Javy klasy `Vector` i `Hashtable` udostępniały bezpieczne wątkowo implementacje tablicy dynamicznej i skrótów. Klasy te są już uważane za przestarzałe i zastąpiono je klasami `ArrayList` i `HashMap`. Klasy te nie są bezpieczne wątkowo. W zamian w bibliotece kolekcji zaproponowano inną technikę. Każdą klasę kolekcji można uczynić bezpieczną wątkowo za pomocą **synchronizacyjnych obiektów opakowujących**:

```
List<E> synchArrayList = Collections.synchronizedList(new ArrayList<E>());
Map<K, V> synchHashMap = Collections.synchronizedMap(new HashMap<K, V>());
```

Metody tak powstałych kolekcji są chronione przez blokadę, co umożliwia bezpieczny wątkowo dostęp.

Należy zapewnić, że żaden wątek nie będzie miał dostępu do struktury danych poprzez oryginalne niesynchronizowane metody. Najprostszym sposobem jest niezapisywanie żadnych referencji do oryginalnego obiektu. Po utworzeniu kolekcji od razu należy przekazać ją do opakowania, tak jak zrobiliśmy to w prezentowanych przykładach.

Nadal trzeba stosować blokowanie po stronie klienta, aby móc *iterować* po kolekcji, podczas gdy inny wątek może ją modyfikować:

```
synchronized (synchHashMap)
{
  Iterator<K> iter = synchHashMap.keySet().iterator();
  while (iter.hasNext()) . . .;
}
```

Tego samego kodu musimy użyć, jeśli korzystamy z pętli typu `for each`, ponieważ pętla ta używa iteratora. Należy pamiętać, że iterator zgłosi wyjątek `ConcurrentModificationException`, jeśli w trakcie iteracji po kolekcji inny wątek ją zmodyfikuje. Synchronizacja jest nadal wymagana, dzięki czemu można wykryć współbieżne modyfikacje.

Zamiast używać synchronizacyjnych obiektów opakowujących, zazwyczaj lepiej jest skorzystać z kolekcji z pakietu `java.util.concurrent`. Słownik `ConcurrentHashMap` został bardzo starannie zaimplementowany w taki sposób, aby można było uzyskać do niego dostęp w wielu wątkach, nie powodując ich wzajemnego blokowania się, pod warunkiem że działają one na różnych komórkach. Jedyny wyjątek stanowi lista tablicowa, która jest często modyfikowana. W takim przypadku synchronizowana lista `ArrayList` może się okazać lepsza od listy `CopyOn`
`↪WriteArrayList`.

`java.util.Collections` **1.2**

- `static <E> Collection<E> synchronizedCollection(Collection<E> c)`
- `static <E> List synchronizedList(List<E> c)`
- `static <E> Set synchronizedSet(Set<E> c)`
- `static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)`
- `static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)`
- `static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)`

  Tworzy widoki kolekcji, których metody są synchronizowane.

12.6. Zadania i pule wątków

Tworzenie nowego wątku jest dość kosztowną operacją, ponieważ wymaga kontaktu z systemem operacyjnym. Jeśli program tworzy dużą liczbę krótkotrwałych wątków, nie należy każdego zadania wykonywać w osobnym wątku, tylko skorzystać z puli wątków. Pula taka zawiera pewną liczbę wątków gotowych do działania. Kiedy programista przekaże do niej obiekt `Runnable`, jeden z wątków wywoła metodę `run`. Gdy zakończy ona działanie, wątek nie zostaje usunięty, tylko jest przechowywany w oczekiwaniu na obsłużenie następnego zadania.

W poniższych sekcjach znajduje się opis narzędzi systemu współbieżności Javy do koordynacji zadań współbieżnych.

12.6.1. Interfejsy Callable i Future

Obiekt implementujący interfejs `Runnable` opakowuje zadania, które działają asynchronicznie. Można go traktować jako asynchroniczną metodę bez parametrów i wartości zwrotnej. Obiekt `Callable` jest podobny do `Runnable`, ale posiada wartość zwrotną. Interfejs `Callable` jest typem parametryzowanym zawierającym jedną metodę o nazwie `call`.

```
public interface Callable<V>
{
  V call() throws Exception;
}
```

Parametr typowy określa typ zwracanej wartości. Na przykład interfejs `Callable<Integer>` reprezentuje asynchroniczne działania, których wynikiem jest obiekt typu `Integer`.

Obiekt `Future` przechowuje *wynik* asynchronicznych obliczeń. Można rozpocząć obliczenia, przekazać gdzieś obiekt `Future` i zapomnieć o nim. Właściciel tego obiektu może pobrać wynik, kiedy będzie gotowy.

W interfejsie `Future<V>` znajdują się następujące metody:

```
V get()
V get(long timeout, TimeUnit unit)
void cancel(boolean mayInterrupt)
boolean isCancelled()
boolean isDone()
```

Wywołanie pierwszej z metod `get` jest zablokowane do zakończenia obliczeń. Druga wersja tej metody również zakłada blokadę, ale zgłasza wyjątek `TimeoutException`, jeśli obliczenia nie zakończą się przed upływem określonego czasu. Obie te metody zgłaszają wyjątek `Interrupted ↪Exception`, jeśli wątek przeprowadzający obliczenia zostanie przerwany. Kiedy obliczenia zakończą się, metoda `get` natychmiast zwraca wartość.

Metoda `isDone` zwraca wartość `false`, jeśli obliczenia są jeszcze w toku, lub `true` w przeciwnym przypadku.

Operację można przerwać za pomocą metody `cancel`. Jeśli jeszcze się nie rozpoczęła, zostanie anulowana i nigdy się nie rozpocznie. Jeśli jest w toku, zostanie przerwana, gdy parametr `mayInterrupt` ma wartość `true`.

Anulowanie zadania to proces dwuetapowy — najpierw należy znaleźć jego wątek, a następnie go przerwać. Implementacja zadania (w metodzie `call`) musi wykryć to przerwanie, po czym zaniechać dalszego wykonywania operacji. Jeśli obiekt `Future` „nie wie", w którym wątku wykonywane jest zadanie, lub jeśli zadanie nie monitoruje stanu przerwania wątku, w którym działa, anulowanie się nie uda.

Jednym ze sposobów na wykonanie obiektu `Callable` jest użycie obiektu `FutureTask`, który implementuje zarówno interfejs `Future`, jak i `Runnable`, dzięki czemu można utworzyć dla niego wątek wykonawczy:

```
Callable<Integer> task = . . .;
var futureTask = new FutureTask<Integer>(task);
var t = new Thread(futureTask); // Runnable
t.start();
. . .
Integer result = task.get();    // Future
```

Częściej obiekt `Callable` przekazuje się do egzekutora. Szerzej piszemy o tym w następnej sekcji.

`java.util.concurrent.Callable<V>` **5.0**

- `V call()`

  Uruchamia zadanie, które zwraca wynik.

`java.util.concurrent.Future<V>` **5.0**

- `V get()`
- `V get(long time, TimeUnit unit)`

  Zwraca wynik, włączając blokadę, dopóki nie jest on dostępny lub nie upłynie określona ilość czasu. Druga wersja zgłasza wyjątek `TimeoutException`, jeśli zakończy się niepowodzeniem.

- `boolean cancel(boolean mayInterrupt)`

  Próbuje anulować wykonywanie zadania. Zadanie, które zostało już uruchomione, a ma parametr `mayInterrupt` ustawiony na `true`, zostanie przerwane. Jeśli operacja anulowania zakończy się pomyślnie, metoda ta zwraca wartość `true`.

- `boolean isCancelled()`

  Zwraca wartość `true`, jeśli zadanie zostało anulowane przed ukończeniem.

- `boolean isDone()`

  Zwraca wartość `true`, jeśli zadanie zostało ukończone w normalny sposób, zostało anulowane lub spowodowało wyjątek.

`java.util.concurrent.FutureTask<V>` **5.0**

- `FutureTask(Callable<V> task)`
- `FutureTask(Runnable task, V result)`

  Tworzy obiekt, który jest zarówno typu `Future<V>`, jak i `Runnable`.

12.6.2. Klasa Executors

W klasie `Executors` znajduje się kilka statycznych metod fabrycznych służących do tworzenia puli wątków (tabela 12.2).

Tabela 12.2. Metody fabryczne klasy Executors

Metoda	Opis
`newCachedThreadPool`	W razie potrzeby tworzy nowe wątki. Nieaktywne wątki są przetrzymywane przez 60 sekund.
`newFixedThreadPool`	Pula zawierająca ustaloną liczbę wątków. Nieaktywne wątki są zachowywane bezterminowo.
`newWorkStealingPool`	Pula odpowiednia do zadań typu „rozgałęzienie-łączenie” (punkt 12.6.4), w ramach których złożone zadanie zostaje podzielone na mniejsze części i nieaktywne wątki „podkradają” prostsze zadania.
`newSingleThreadExecutor`	Pula składająca się z jednego wątku wykonującego zadania po kolei (podobnie do wątku dystrybucji zdarzeń w Swing).
`newScheduledThreadPool`	Ustalona harmonogramowana pula wątków. Zastępstwo dla `java.util.Timer`.
`newSingleThreadScheduledExecutor`	Harmonogramowana pula składająca się z jednego wątku.

Metoda `newCachedThreadPool` tworzy pulę wątków, która wykonuje zadania natychmiast za pomocą jednego z wątków nieaktywnych, jeśli taki jest, lub tworząc nowy wątek w przeciwnym przypadku. Metoda `newFixedThreadPool` tworzy pulę wątków o ustalonym rozmiarze. Jeśli zadań jest więcej niż wolnych wątków, są one umieszczane w kolejce i wykonywane po zakończeniu wcześniejszych zadań. Metoda `newSingleThreadExecutor` tworzy pulę składającą się z jednego wątku, który wykonuje zadania jedno po drugim. Wszystkie trzy opisane metody zwracają obiekt klasy `ThreadPoolExecutor`, która implementuje interfejs `Executor`
`↪Service`.

Buforowanej puli wątków należy używać, gdy korzysta się z krótkotrwałych wątków lub dużo czasu traci się na blokowanie. Jeśli jednak wątki intensywnie pracują bez blokowania, lepiej nie uruchamiać zbyt wielu naraz.

Optymalną szybkość wykonywania uzyskuje się przy liczbie współbieżnych wątków odpowiadającej liczbie rdzeni procesora. W takiej sytuacji należy utworzyć stałą pulę wątków.

Jednowątkowy egzekutor umożliwia analizę wydajności. Jeśli tymczasowo zamieni się buforowaną lub stałą pulę wątków na pulę jednowątkową, można sprawdzić, w jakim stopniu brak współbieżności spowolni działanie aplikacji.

W Javie EE dostępna jest podklasa `ManagedExecutorService` odpowiednia do pracy z zadaniami współbieżnymi w środowisku Java EE. Także systemy sieciowe, takie jak Play, zawierają usługi wykonawcze przeznaczone dla zadań w obrębie systemu.

Obiekt Runnable lub Callable można przekazać do ExecutorService za pomocą jednej z poniższych metod:

```
Future<T> submit(Callable<T> task)
Future<?> submit(Runnable task)
Future<T> submit(Runnable task, T result)
```

Pula wykona powierzone jej zadanie przy najbliższej sposobności. Metoda submit zwraca obiekt typu Future, przy użyciu którego można pobrać wynik lub anulować zadanie.

Druga z wymienionych metod zwraca dość osobliwie wyglądający typ Future<?>. Na rzecz tego obiektu można wywołać metody isDone, cancel lub isCancelled. Natomiast metoda get w chwili ukończenia zwraca wartość null.

Trzecia wersja metody submit zwraca obiekt Future, którego metoda get zwraca wynik operacji, gdy jest już gotowy.

Po zakończeniu pracy w puli wątków należy wywołać metodę shutdown. Inicjalizuje ona operację zamykającą pulę. Egzekutor, który jest zamykany, nie przyjmuje żadnych nowych zadań. Po zakończeniu wszystkich zadań wątki puli zostają zakończone. Istnieje także metoda shutdownNow, która powoduje, że pula anuluje wszystkie jeszcze niezaczęte zadania.

Oto zestawienie działań, które należy wykonać, aby użyć puli wątków:

1. Wywołaj statyczną metodę newCachedThreadPool lub newFixedThreadPool z klasy Executors.
2. Przekaż obiekty Runnable lub Callable za pomocą metody submit.
3. Wykorzystaj zwrócone obiekty Future, jeśli chcesz mieć możliwość anulowania zadań lub jeśli przekażesz obiekty Callable.
4. Jeśli nie chcesz przekazywać więcej zadań, wywołaj metodę shutdown.

Interfejs ScheduledExecutorService zawiera metody służące do planowego i wielokrotnego wykonywania zadań. Tworzenie pul wątków jest możliwe dzięki uogólnieniu klasy java.util.
↪Timer. Metody newScheduledThreadPool i newSingleThreadScheduledExecutor klasy Executors zwracają obiekty implementujące interfejs ScheduledExecutorService.

Zadania Runnable i Callable można zaplanować do jednorazowego wykonania po uprzednim odłożeniu ich na jakiś czas. Zadanie Runnable można także wykonywać w określonych odstępach czasu. Szczegółowe informacje na ten temat znajdują się w wyciągach z API.

java.util.concurrent.Executors **5.0**

- ExecutorService newCachedThreadPool()

  Zwraca pulę wątków, która w razie potrzeby tworzy wątki, i kończy te, które są nieaktywne przez 60 sekund.

- ExecutorService newFixedThreadPool(int threads)

  Zwraca pulę wątków, która wykonuje zadania przy użyciu określonej liczby wątków.

- `ExecutorService newSingleThreadExecutor()`

  Zwraca egzekutor, który wykonuje zadania kolejno w jednym wątku.

- `ScheduledExecutorService newScheduledThreadPool(int threads)`

  Zwraca pulę wątków wykorzystującą określoną liczbę wątków do planowania zadań.

- `ScheduledExecutorService newSingleThreadScheduledExecutor()`

  Zwraca egzekutor planujący zadania w jednym wątku.

`java.util.concurrent.ExecutorService` **5.0**

- `Future<T> submit(Callable<T> task)`
- `Future<T> submit(Runnable task, T result)`
- `Future<?> submit(Runnable task)`

  Przekazuje zadanie do wykonania.

- `void shutdown()`

  Zamyka usługę. Kończy przekazane wcześniej zadania, ale nie przyjmuje nowych.

`java.util.concurrent.ThreadPoolExecutor` **5.0**

- `int getLargestPoolSize()`

  Zwraca największy rozmiar puli wątków w czasie działania egzekutora.

`java.util.concurrent.ScheduledExecutorService` **5.0**

- `ScheduledFuture<V> schedule(Callable<V> task, long time, TimeUnit unit)`
- `ScheduledFuture<?> schedule(Runnable task, long time, TimeUnit unit)`

  Wykonuje dane zadanie po upływie określonej ilości czasu.

- `ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit)`

  Ustawia harmonogram uruchamiania danego zadania w równych odstępach czasu, zaczynając po upływie początkowego opóźnienia (`initialDelay`).

- `ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)`

  Ustawia harmonogram uruchamiania danego zadania w określonych odstępach czasu. Długość czasu opóźnienia kolejnych wykonań wynosi tyle, ile upłynęło czasu od ukończenia jednego wywołania do rozpoczęcia kolejnego. Pierwsze wykonanie następuje po upływie `initialDelay` czasu.

12.6.3. Kontrolowanie grup zadań

Wiemy już, jak wykorzystywać egzekutor w roli puli wątków mającej na celu zwiększenie szybkości wykonywania zadań. Czasami egzekutory są wykorzystywane do bardziej taktycznych celów, na przykład kontrolowania grup spokrewnionych zadań. Na przykład wszystkie zadania w egzekutorze można zakończyć za pomocą jednego wywołania metody `shutdownNow`.

Metoda `invokeAny` przekazuje wszystkie obiekty z kolekcji obiektów typu `Callable` i zwraca wynik ukończonego zadania. Nie wiadomo, które to zadanie — prawdopodobnie to, które zostało ukończone najwcześniej. Metody tej można użyć w algorytmie wyszukującym, który może przyjąć każde rozwiązanie. Wyobraźmy sobie na przykład, że chcemy rozłożyć na czynniki dużą liczbę całkowitą — operacja wymagana do łamania szyfru RSA. Można przekazać kilka zadań, z których każde próbuje rozkładu przy użyciu liczb z innego zakresu. Kiedy tylko którekolwiek z nich ma odpowiedź, dalsze obliczenia można zatrzymać.

Metoda `invokeAll` przesyła wszystkie obiekty `Callable` z kolekcji i zwraca listę obiektów `Future`, które reprezentują rozwiązania wszystkich zadań. Wyniki te można przetwarzać w następujący sposób:

```
List<Callable<T>> tasks = . . .;
List<Future<T>> results = executor.invokeAll(tasks);
for (Future<T> result : results)
  processFurther(result.get());
```

W pętli `for` pierwsze wywołanie metody `result.get()` włącza blokadę, aż pierwszy wynik stanie się dostępny. Nie stanowi to problemu, jeśli wszystkie zadania kończą się mniej więcej w tym samym czasie. Jednak lepiej byłoby pobierać wyniki w takiej kolejności, w jakiej są udostępniane. Można to osiągnąć za pomocą klasy `ExecutorCompletionService`.

Należy zacząć od utworzenia w normalny sposób egzekutora. Następnie tworzymy obiekt `ExecutorCompletionService`, do którego przekazujemy zadania. Obiekt ten zarządza kolejką blokującą zawierającą obiekty typu `Future`, w których zapisywane są wyniki zadań. W związku z tym powyższe obliczenia można wykonać szybciej za pomocą poniższego algorytmu:

```
var service = new ExecutorCompletionService<T>(executor);
ExecutorCompletionService(executor);
for (Callable<T> task : tasks) service.submit(task);
for (int i = 0; i < tasks.size(); i++)
  processFurther(service.take().get());
```

Program na listingu 12.8 przedstawia przykład użycia obiektów `Callable` i egzekutorów. Na początku sprawdzamy, ile plików w drzewie katalogów zawiera dane słowo. Dla każdego pliku tworzymy osobne zadanie:

```
Set<Path> files = descendants(Path.of(start));
var tasks = new ArrayList<Callable<Long>>();
for (Path file : files)
{
    Callable<Long> task = () -> occurrences(word, file);
    tasks.add(task);
}
```

Następnie przekazujemy te zadania do usługi egzekutora:

```
ExecutorService executor = Executors.newCachedThreadPool();
List<Future<Long>> results = executor.invokeAll(tasks);
```

Aby otrzymać ogólny wynik, sumujemy wszystkie wyniki, włączając blokadę do czasu, aż staną się dostępne:

```
long total = 0;
for (Future<Long> result : results)
total += result.get();
```

Ponadto program ten pokazuje czas spędzony na każdym poszukiwaniu. Wypakuj kod źródłowy JDK i włącz wyszukiwanie. Następnie zamień usługę egzekutora na egzekutor jednowątkowy i włącz wyszukiwanie ponownie, aby sprawdzić, czy wersja współbieżna była szybsza.

W drugiej części programu szukamy pierwszego pliku zawierającego dane słowo. Wyszukiwanie równoległe realizujemy za pomocą metody `invokeAny`. W tym przypadku musimy uważniej zdefiniować zadania. Metoda `invokeAny` kończy działanie natychmiast, gdy któreś z zadań zwróci wynik. W związku z tym nie ma możliwości zdefiniowania zadań wyszukiwania, które zwracałyby wartość logiczną oznaczającą sukces lub porażkę. Nie chcemy kończyć wyszukiwania, gdy jakieś zadanie zakończy się niepowodzeniem. Zakończone niepowodzeniem zadanie zgłasza wyjątek `NoSuchElementException`. Ponadto pomyślne zakończenie jednego zadania powoduje anulowanie pozostałych zadań. Dlatego monitorujemy stan przerwania. Jeśli wątek zostanie przerwany, zadanie wyszukiwania przed zamknięciem drukuje informację, dzięki czemu wiadomo, że operacja anulowania zadziałała.

```
public static Callable<Path> searchForTask(String word, Path path)
{
    return () -> {
        try (var in = new Scanner(path))
        {
            while (in.hasNext())
            {
                if (in.next().equals(word)) return path;
                if (Thread.currentThread().isInterrupted())
                {
                    System.out.println("Szukanie w " + path + " anulowano.");
                    return null;
                }
            }
            throw new NoSuchElementException();
        }
    };
}
```

Ten program drukuje rozmiar największej puli, jaka powstała podczas wykonywania. Nie da się tego sprawdzić poprzez interfejs `ExecutorService`. Dlatego konieczne było wykonanie rzutowania obiektu puli na klasę `ThreadPoolExecutor`.

Patrząc na kod tego programu, można docenić praktyczność usług egzekutorów. We własnych programach staraj się używać egzekutorów do zarządzania wątkami, zamiast uruchamiać wątki indywidualnie.

Listing 12.8. executors/ExecutorDemo.java

```
package executors;

import java.io.*;
import java.nio.file.*;
import java.time.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;

/**
 * Program demonstrujący interfejs Callable i egzekutory.
 * @version 1.0 2018-01-04
 * @author Cay Horstmann
 */
public class ExecutorDemo
{
   /**
    * Liczy wystąpienia danego słowa w pliku.
    * @return liczba wystąpień danego słowa
    */
   public static long occurrences(String word, Path path)
   {
      try (var in = new Scanner(path))
      {
         int count = 0;
         while (in.hasNext())
            if (in.next().equals(word)) count++;
         return count;
      }
      catch (IOException ex)
      {
         return 0;
      }
   }

   /**
    * Zwraca wszystkie podkatalogi danego katalogu — zobacz rozdziały 1. i 2. w tomie II.
    * @param rootDir katalog główny
    * @return zbiór wszystkich podkatalogów katalogu głównego
    */
   public static Set<Path> descendants(Path rootDir) throws IOException
   {
      try (Stream<Path> entries = Files.walk(rootDir))
      {
         return entries.filter(Files::isRegularFile)
            .collect(Collectors.toSet());
      }
   }

   /**
    * Tworzy zadanie szukające słowa w pliku.
    * @param word szukane słowo
    * @param path plik do przeszukania
    * @return zadanie wyszukiwania, które zwraca ścieżkę w przypadku powodzenia
    */
   public static Callable<Path> searchForTask(String word, Path path)
   {
```

```
      return () -> {
         try (var in = new Scanner(path))
         {
            while (in.hasNext())
            {
               if (in.next().equals(word)) return path;
               if (Thread.currentThread().isInterrupted())
               {
                  System.out.println("Szukanie w " + path + " anulowano.");
                  return null;
               }
            }
            throw new NoSuchElementException();
         }
      };
   }

   public static void main(String[] args)
         throws InterruptedException, ExecutionException, IOException
   {
      try (var in = new Scanner(System.in))
      {
         System.out.print("Wpisz ścieżkę do katalogu podstawowego (np. /opt/jdk-9-src): ");
         String start = in.nextLine();
         System.out.print("Wpisz słowo kluczowe (np. volatile): ");
         String word = in.nextLine();

         Set<Path> files = descendants(Path.of(start));
         var tasks = new ArrayList<Callable<Long>>();
         for (Path file : files)
         {
            Callable<Long> task = () -> occurrences(word, file);
            tasks.add(task);
         }
         ExecutorService executor = Executors.newCachedThreadPool();
         // użyj egzekutora wątku, aby sprawdzić, czy
         // większa liczba wątków przyspiesza wyszukiwanie
         // ExecutorService executor = Executors.newSingleThreadExecutor();

         Instant startTime = Instant.now();
         List<Future<Long>> results = executor.invokeAll(tasks);
         long total = 0;
         for (Future<Long> result : results)
            total += result.get();
         Instant endTime = Instant.now();
         System.out.println("Liczba wystąpień słowa " + word + ": " + total);
         System.out.println("Czas: "
            + Duration.between(startTime, endTime).toMillis() + " ms");

         var searchTasks = new ArrayList<Callable<Path>>();
         for (Path file : files)
            searchTasks.add(searchForTask(word, file));
         Path found = executor.invokeAny(searchTasks);
         System.out.println(word + " występuje w: " + found);

         if (executor instanceof ThreadPoolExecutor) // egzekutor jednowątkowy nie jest
            System.out.println("Największy rozmiar puli: "
               + ((ThreadPoolExecutor) executor).getLargestPoolSize());
```

```
            executor.shutdown();
         }
      }
   }
```

`java.util.concurrent.ExecutorService` **5.0**

- `T invokeAny(Collection<Callable<T>> tasks)`
- `T invokeAny(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)`

  Wykonuje podane zadania i zwraca wynik jednego z nich. Druga z tych metod zgłasza wyjątek `TimeoutException`, jeśli zostanie przekroczony dozwolony czas.
- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks)`
- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)`

  Wykonuje dane zadania i zwraca wyniki ich wszystkich. Druga z tych metod zgłasza wyjątek `TimeoutException`, jeśli zostanie przekroczony dozwolony czas.

`java.util.concurrent.ExecutorCompletionService` **5.0**

- `ExecutorCompletionService(Executor e)`

  Tworzy obiekt typu `ExecutorCompletionService`, który przechowuje wyniki zadań określonego egzekutora.
- `Future<V> submit(Callable<V> task)`
- `Future<V> submit(Runnable task, V result)`

  Przekazuje zadanie do egzekutora.
- `Future<V> take()`

  Usuwa następny wynik lub włącza blokadę, jeśli nie ma dostępnych żadnych wyników.
- `Future<V> poll()`
- `Future<V> poll(long time, TimeUnit unit)`

  Usuwa następny wynik lub zwraca wartość `null`, jeśli nie ma dostępnych żadnych wyników. Druga wersja tej metody odczekuje określoną ilość czasu.

12.6.4. Metoda rozgałęzienie-złączenie

W niektórych aplikacjach używanych jest wiele wątków, z których większość jest nieaktywna. Przykładem jest serwer sieciowy obsługujący każde połączenie w osobnym wątku. Są też aplikacje tworzące po jednym wątku dla każdego rdzenia procesora. Robią tak choćby aplikacje wykonujące wymagające obliczenia, np. przy przetwarzaniu grafiki albo filmów. Metoda rozgałęzienie-złączenie (ang. *fork-join*), która powstała w Java SE 7, służy do rozwiązywania problemów dotyczących drugiego z wymienionych przypadków. Wyobraźmy sobie, że mamy zadanie przetwarzania, które można naturalnie rozłożyć na podzadania:

```
if (rozmiarProblemu < prog)
    rozwiąż problem bezpośrednio
else
{
    podziel problem na części
    rekurencyjnie wykonaj każdą część
    połącz wyniki
}
```

Jednym z przykładów jest przetwarzanie obrazu. Obraz można poprawić, przekształcając jego górną i dolną połowę. Jeśli ma się do dyspozycji wystarczającą liczbę wolnych procesorów, operacje te można wykonywać równocześnie (trzeba będzie dodatkowo wykonać pracę związaną z połączeniem połówek, ale to mało istotny szczegół).

My przeanalizujemy prostszy przykład. Przypuśćmy, że chcemy się dowiedzieć, ile elementów tablicy spełnia pewien warunek. Dzielimy ją na pół, wykonujemy obliczenia dla każdej z połówek osobno, a następnie sumujemy wyniki.

Aby wykonać nasze rekurencyjne obliczenia w odpowiedni sposób, utworzymy klasę rozszerzającą klasę `RecursiveTask<T>` (jeśli wynik obliczenia jest typu `T`) lub `RecursiveAction` (jeśli nie ma wyniku). Przesłonimy metodę `compute`, aby generowała i wywoływała części zadania oraz łączyła ich wyniki.

```
class Counter extends RecursiveTask<Integer>
{
    . . .
    protected Integer compute()
    {
        if (to - from < THRESHOLD)
        {
            bezpośrednie rozwiązanie problemu
        }
        else
        {
            int mid = (from + to) / 2;
            var first = new Counter(values, from, mid, filter);
            var second = new Counter(values, mid, to, filter);
            invokeAll(first, second);
            return first.join() + second.join();
        }
    }
}
```

Metoda `invokeAll` otrzymuje liczbę zadań i włącza blokadę, dopóki wszystkie one nie zostaną wykonane. Metoda `join` generuje wynik. Stosujemy ją do wszystkich podzadań, aby otrzymać sumę.

Istnieje też metoda `get` do pobierania aktualnego wyniku, ale jest ona mniej atrakcyjna, ponieważ może zgłaszać kontrolowane wyjątki, których nie możemy ponownie zgłaszać w metodzie `compute`.

Pełny kod źródłowy przykładu jest przedstawiony na listingu 12.9.

Pule typu rozgałęzienie-złączenie są zoptymalizowane pod kątem pracy bez blokowania. Jeśli do takiej puli zostanie dodanych zbyt wiele blokujących zadań, może dojść do jej zagłodzenia. Problem ten można rozwiązać przez implementację interfejsu `ForkJoinPool.ManagedBlocker`, ale to zaawansowana technika, której nie będziemy opisywać.

Listing 12.9. forkJoin/forkJoinTest.java

```
package forkJoin;

import java.util.concurrent.*;
import java.util.function.*;

/**
 * Program demonstrujący technikę rozgałęzienie-złączenie
 * @version 1.01 2015-06-21
 * @author Cay Horstmann
 */
public class ForkJoinTest
{
   public static void main(String[] args)
   {
      final int SIZE = 10000000;
      var numbers = new double[SIZE];
      for (int i = 0; i < SIZE; i++) numbers[i] = Math.random();
      var counter = new Counter(numbers, 0, numbers.length, x -> x > 0.5);
      var pool = new ForkJoinPool();
      pool.invoke(counter);
      System.out.println(counter.join());
   }
}

class Counter extends RecursiveTask<Integer>
{
   public static final int THRESHOLD = 1000;
   private double[] values;
   private int from;
   private int to;
   private DoublePredicate filter;

   public Counter(double[] values, int from, int to, DoublePredicate filter)
   {
      this.values = values;
      this.from = from;
      this.to = to;
      this.filter = filter;
   }

   protected Integer compute()
   {
      if (to - from < THRESHOLD)
      {
         int count = 0;
         for (int i = from; i < to; i++)
         {
            if (filter.test(values[i])) count++;
         }
         return count;
```

```
        }
        else
        {
           int mid = (from + to) / 2;
           var first = new Counter(values, from, mid, filter);
           var second = new Counter(values, mid, to, filter);
           invokeAll(first, second);
           return first.join() + second.join();
        }
     }
  }
```

Technika rozgałęzienie-złączenie wykorzystuje efektywny algorytm heurystyczny pozwalający zrównoważyć obciążenie poszczególnych wątków, o nazwie **podkradanie pracy** (ang. *work stealing*). Każdy wątek roboczy ma kolejkę dwukierunkową dla zadań i podzadania umieszcza na jej początku (tylko jeden wątek ma dostęp do tego miejsca, więc nie ma potrzeby stosować blokady). Gdy wątek jest nieaktywny, „podkrada" zadanie z końca innej kolejki dwukierunkowej. Jako że duże podzadania są w ogonie, do podkradania dochodzi rzadko.

12.7. Obliczenia asynchroniczne

Do tej pory obliczenia współbieżne wykonywaliśmy w ten sposób, że dzieliliśmy zadanie na części, a następnie czekaliśmy na ich wykonanie. Tylko że czekanie nie zawsze jest najlepszym pomysłem. W tym podrozdziale nauczysz się implementować obliczenia niewymagające oczekiwania, czyli **asynchroniczne**.

12.7.1. Klasa CompletableFuture

Kiedy dany jest obiekt `Future`, należy wywołać metodę `get`, która włącza blokadę, aby uzyskać wartość. Klasa `CompletableFuture` implementuje interfejs `Future` i zapewnia drugi mechanizm uzyskiwania wyniku. Należy zarejestrować **metodę zwrotną**, która zostanie wywołana (w jakimś wątku) z wynikiem, gdy ten stanie się dostępny.

```
CompletableFuture<String> f = . . .;
f.thenAccept(s -> Process the result string s);
```

W ten sposób można przetwarzać wynik bez stosowania blokady, gdy stanie się dostępny.

W API jest kilka metod zwracających obiekty `CompletableFuture`. Można na przykład asynchronicznie pobierać stronę internetową za pomocą eksperymentalnej klasy `HttpClient`, której opis znajduje się w rozdziale 4. tomu II:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder(URI.create(urlString)).GET().build();
CompletableFuture<HttpResponse<String>> f = client.sendAsync(
request, BodyHandler.asString());
```

Jeśli istnieje metoda tworząca gotowy obiekt `CompletableFuture`, to mamy szczęście, ale w większości przypadków konieczne jest utworzenie jej własnoręcznie. Aby wykonać zadanie asynchronicznie i uzyskać obiekt `CompletableFuture`, nie należy go przekazywać bezpośrednio do usługi egzekutora. W zamian należy wywołać statyczną metodę `CompletableFuture.supplyAsync`. Poniżej znajduje się przykładowy program wczytujący stronę internetową bez użycia klasy `HttpClient`:

```
public CompletableFuture<String> readPage(URL url)
{
    return CompletableFuture.supplyAsync(() ->
    {
        try
        {
            return new String(url.openStream().readAllBytes(), "UTF-8");
        }
        catch (IOException e)
        {
            throw new UncheckedIOException(e);
        }
    }, executor);
}
```

Jeśli nie wskażesz egzekutora, zadanie zostanie wykonane na domyślnym egzekutorze (a konkretnie na tym, który zwróci metoda `ForkJoinPool.commonPool()`). Zazwyczaj nie jest to pożądane.

Pierwszy argument metody `supplyAsync` to `Supplier<T>`, a nie `Callable<T>`. Oba interfejsy opisują funkcje bez argumentów i wartość zwrotną typu `T`, ale funkcja `Supplier` nie może zgłaszać wyjątków kontrolowanych. Jak widać w powyższym kodzie, nie był to szczęśliwy wybór.

`CompletableFuture` może zakończyć działanie na dwa sposoby: zwracając wynik lub zgłaszając nieprzechwycony wyjątek. W obu przypadkach należy użyć metody `whenComplete`. Dostarczona funkcja zostaje wywołana z wynikiem (lub `null`, jeśli brak wyniku) i wyjątkiem (lub `null`, jeśli brak wyjątku).

```
f.whenComplete((s, t) -> {
    if (t == null) { Przetwarzanie wyniku s; }
    else { Przetwarzanie obiektu Throwable t; }
});
```

Klasa `CompletableFuture` pozwala na ręczne ustawienie wartości ukończenia. (W innych bibliotekach współbieżności taki obiekt nazywa się obietnicą). Oczywiście, gdy obiekt `CompletableFuture` zostanie utworzony za pomocą metody `supplyAsync`, wartość ukończenia zostaje niejawnie ustawiona w chwili zakończenia zadania. Natomiast jawne ustawienie wartości wyniku daje programiście dodatkową elastyczność. Na przykład, dwa zadania mogą jednocześnie obliczać odpowiedź:

```
var f = new CompletableFuture<Integer>();
executor.execute(() ->
    {
        int n = workHard(arg);
        f.complete(n);
```

```
    });
executor.execute(() ->
    {
        int n = workSmart(arg);
        f.complete(n);
    });
```

Aby natomiast uzyskać zakończenie wyjątkiem, należy zastosować następujące wywołanie:

```
Throwable t = . . .;
f.completeExceptionally(t);
```

Wywołanie metod `complete` lub `completeExceptionally` na tym samym obiekcie `Future` w różnych wątkach jest bezpieczne. Jeśli obiekt jest już ukończony, wywołanie nie powoduje żadnego efektu.

Metoda `isDone` pozwala sprawdzić, czy obiekt `Future` został ukończony (normalnie lub z wyjątkiem). W poprzednim przykładzie metody `workHard` i `workSmart` mogą wykorzystać tę informację w celu zakończenia pracy w chwili znalezienia wyniku przez drugą z nich.

W odróżnieniu od zwykłych obiektów `Future`, obliczenia obiektu `CompletableFuture` nie zostają przerwane w chwili wywołania metody `cancel`. Anulowanie stanowi jedynie sygnał do tego, że obiekt `Future` ma zostać ukończony ze zgłoszeniem wyjątku `CancellationException`. Generalnie jest to logiczne, ponieważ obiekt `CompletableFuture` nie może mieć pojedynczego wątku odpowiedzialnego za jego ukończenie. Ograniczenie to dotyczy także obiektów `CompletableFuture` zwracanych przez takie metody, jak `supply` ↪`Async`, które z zasady można przerywać.

12.7.2. Tworzenie obiektów CompletableFuture

Wywołania nieblokujące są implementowane poprzez wywołania zwrotne. Programista rejestruje wywołanie zwrotne dla czynności, która ma zostać wykonana po tym, jak dane zadanie zostanie ukończone. Oczywiście, jeśli kolejna czynność także jest asynchroniczna, następna czynność po tym jest w innym wywołaniu zwrotnym. Choć programista może myśleć, że najpierw ma zostać wykonany krok 1., potem 2., a następnie 3., logika programu może zostać „piekielnie" rozproszona. Sprawa komplikuje się jeszcze bardziej, gdy trzeba dodać obsługę błędów. Powiedzmy, że krok 2. to logowanie użytkownika. Może być konieczne jego powtórzenie, ponieważ istnieje ryzyko, że użytkownik źle wpisze swoją nazwę albo hasło. Implementacja takiej logiki w zestawie wywołań zwrotnych i zrozumienie jej, gdy zostanie napisana przez kogoś innego, to duże wyzwanie.

Klasa `CompletableFuture` rozwiązuje ten problem, umożliwiając tworzenie potoków przetwarzania zadań asynchronicznych.

Powiedzmy na przykład, że chcemy pobrać wszystkie obrazy ze strony internetowej. Mamy taką metodę:

```
public void CompletableFuture<String> readPage(URL url)
```

Metoda ta zwraca tekst ze strony, gdy ten stanie się dostępny. Jeżeli metoda:

```
public List<URL> getImageURLs(String page)
```

zwraca adresy URL obrazów znalezionych na stronie HTML, to można zaplanować jej wywołanie, gdy strona będzie dostępna:

```
CompletableFuture<String> contents = readPage(url);
CompletableFuture<List<URL>> imageURLs = contents.thenApply(this::getLinks);
```

Metoda `thenApply` także nie jest metodą blokującą. Zwraca kolejny obiekt `Future`. Jeśli pierwszy zakończył działanie, jego wynik jest przekazywany do metody `getImageURLs` i wartość zwrotna tej metody staje się ostatecznym wynikiem.

Przy użyciu obiektów klasy `CompletableFuture` można tylko określić, co i w jakiej kolejności ma zostać zrobione. Oczywiście nie wszystko zostanie wykonane naraz, ale najważniejsze jest to, że cały kod mamy w jednym miejscu.

Teoretycznie `CompletableFuture` jest prostym API, ale istnieje wiele wariantów metod do składania obiektów tego typu. Najpierw przyjrzymy się tym, które pracują na pojedynczym obiekcie (tabela 12.3). (Każdej przedstawionej metodzie towarzyszą dwa warianty `Async`, których tu nie opisano. Jeden z nich wykorzystuje wspólny parametr `ForkJoinPool`, a drugi — `Executor`). W tabeli skrótowo zapisuję rozwlekłe interfejsy funkcyjne, tzn. zamiast `Function<? super T, U>` piszę `T -> U`. To oczywiście nie są prawdziwe typy Javy.

Wiesz już, jak wywoływać metodę `thenApply`. Wyobraź sobie, że `f` jest funkcją, która pobiera wartości typu `T` i zwraca wartości typu `U`. Spójrz na poniższe wywołania:

```
CompletableFuture<U> future.thenApply(f);
CompletableFuture<U> future.thenApplyAsync(f);
```

Pierwsze z tych wywołań zwraca obiekt `Future` stosujący `f` do wyniku `future`, gdy ten jest dostępny. Drugie natomiast wywołuje `f` w innym wątku.

Metoda `thenCompose`, zamiast pobierać funkcję mapującą typ `T` na typ `U`, pobiera funkcję mapującą `T` na `CompletableFuture<U>`. Brzmi to dość abstrakcyjnie, ale może być całkiem naturalne. Pomyśl o czynności odczytania strony internetowej znajdującej się pod podanym adresem URL. Zamiast przekazywać metodę:

```
public String blockingReadPage(URL url)
```

bardziej eleganckim rozwiązaniem jest sprawienie, by metoda ta zwracała obiekt `Future`:

```
public CompletableFuture<String> readPage(URL url)
```

Teraz wyobraź sobie, że masz inną metodę odbierającą adres URL od użytkownika, np. z okna dialogowego, które nie przekaże odpowiedzi, dopóki użytkownik nie kliknie przycisku *OK*. To także jest przyszłe zdarzenie:

```
public CompletableFuture<URL> getURLInput(String prompt)
```

Mamy tu dwie funkcje: `T -> CompletableFuture<U>` i `U -> CompletableFuture<V>`. Jeśli druga z nich zostanie wywołana po zakończeniu działania przez pierwszą, to komponują się one do postaci `T -> CompletableFuture<V>`. Właśnie to robi metoda `thenCompose`.

W poprzedniej sekcji pokazaliśmy, jak obsługiwać wyjątki za pomocą metody `whenComplete`. Istniej też metoda `handle`, która wymaga funkcji przetwarzającej wynik lub wyjątek i obliczającej nowy wynik. W wielu przypadkach prościej jest wywołać metodę `exceptionally`, która w przypadku wystąpienia wyjątku oblicza wartość zastępczą:

```
CompletableFuture<List<URL>> imageURLs = readPage(url)
    .exceptionally(ex -> "<html></html>")
    .thenApply(this::getImageURLs)
```

W taki sam sposób można postąpić w przypadku limitu czasu:

```
CompletableFuture<List<URL>> imageURLs = readPage(url)
    .completeOnTimeout("<html></html>", 30, TimeUnit.SECONDS)
    .thenApply(this::getImageURLs)
```

Ewentualnie w chwili upływu czasu można zgłosić wyjątek:

```
CompletableFuture<String> = readPage(url).orTimeout(30, TimeUnit.SECONDS)
```

Metody w tabeli 12.3 z wynikiem `void` są używane na końcu potoku przetwarzania.

Tabela 12.3. Dodawanie akcji do obiektu typu CompletableFuture<T>

Metoda	Parametr	Opis
`thenApply`	`T -> U`	Wywołuje funkcję na wyniku.
`thenAccept`	`T -> void`	To samo co `thenApply`, tylko z wynikiem `void`.
`thenCompose`	`T -> CompletableFuture<U>`	Wywołuje funkcję na wyniku i wykonuje zwrócony obiekt `Future`.
`handle`	`(T, Throwable) -> U`	Przetwarza wynik lub błąd i nowy wynik.
`whenComplete`	`(T, Throwable) -> void`	To samo co `handle`, tylko z wynikiem `void`.
`exceptionally`	`Throwable -> T`	Oblicza wynik z błędu.
`completeOn` `↪Timeout`	`T, long, TimeUnit`	Zwraca daną wartość jako wynik w przypadku upływu limitu czasu.
`orTimeout`	`long, TimeUnit`	Zwraca `TimeoutException` w przypadku przekroczenia limitu czasu.
`thenRun`	`Runnable`	Wykonuje `Runnable` z wynikiem `void`.

Teraz przyjrzymy się metodom tworzącym kombinacje obiektów `Future` (tabela 12.4).

Trzy pierwsze z tych metod wykonują czynności `CompletableFuture<T>` i `CompletableFuture<U>` współbieżnie, a następnie łączą wyniki.

Następne trzy metody wykonują dwie czynności `CompletableFuture<T>` współbieżnie. Gdy jedna z nich zostanie zakończona, jej wynik zostaje przekazany dalej, a wynik drugiej jest ignorowany.

Statyczne metody `allOf` i `anyOf` pobierają dowolną liczbę obiektów `CompletableFuture` i zwracają `CompletableFuture<Void>`, który kończy działanie, gdy wszystkie z nich lub którykolwiek z nich zakończy działanie. Metoda `allOf` nie zwraca wyniku, a metoda `anyOf` nie kończy pozostałych zadań.

Tabela 12.4. Tworzenie kombinacji obiektów Future

Metoda	Parametr	Opis
thenCombine	CompletableFuture<U>, (T, U) -> V	Wykonuje obie czynności i łączy wyniki za pomocą podanej funkcji.
thenAcceptBoth	CompletableFuture<U>, (T, U) -> void	To samo co thenCombine, tylko z wynikiem void.
runAfterBoth	CompletableFuture<?>, Runnable	Wykonuje Runnable po zakończeniu obu czynności.
applyToEither	CompletableFuture<T>, T -> V	Jeśli dostępny jest wynik jednej lub drugiej czynności, przekazuje go do podanej funkcji.
acceptEither	CompletableFuture<T>, T -> void	To samo co applyToEither, tylko z wynikiem void.
runAfterEither	CompletableFuture<?>, Runnable	Wykonuje Runnable po zakończeniu jednej lub drugiej czynności.
static allOf	CompletableFuture<?>...	Kończy z wynikiem void po tym, jak wszystkie dane obiekty Future zakończą działanie.
static anyOf	CompletableFuture<?>...	Kończy z wynikiem void po tym, jak którykolwiek z danych obiektów Future zakończy działanie.

W rzeczywistości metody opisane w tej sekcji przyjmują parametry typu CompletionStage, a nie CompletableFuture. Interfejs CompletionStage opisuje sposób komponowania obliczeń asynchronicznych, podczas gdy interfejs Future koncentruje się na wyniku obliczeń. CompletableFuture to połączenie CompletionStage i Future.

Listing 12.10 przedstawia kompletny program wczytujący stronę internetową, szukający na niej obrazów, ładujący te obrazy i zapisujący je na dysku lokalnym. Zwróć uwagę, że wszystkie czasochłonne metody zwracają CompletableFuture. W celu uruchomienia obliczeń asynchronicznych stosujemy pewną sztuczkę. Zamiast bezpośrednio wywołać metodę readPage, tworzymy ukończony obiekt Future z argumentem w postaci adresu URL, po czym kombinujemy go z this::readPage. Dzięki temu potok ma jednolity wygląd:

```
CompletableFuture.completedFuture(url)
    .thenComposeAsync(this::readPage, executor)
    .thenApply(this::getImageURLs)
    .thenCompose(this::getImages)
    .thenAccept(this::saveImages);
```

Listing 12.10. completableFutures/CompletableFutureDemo.java

```
package completableFutures;

import java.awt.image.*;
import java.io.*;
import java.net.*;
import java.nio.charset.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.regex.*;
```

```
import javax.imageio.*;

public class CompletableFutureDemo
{
   private static final Pattern IMG_PATTERN = Pattern.compile(

"[<]\\s*[iI][mM][gG]\\s*[^>]*[sS][rR][cC]\\s*[=]\\s*['\"]([^'\"]*)['\"][^>]*[>]");
   private ExecutorService executor = Executors.newCachedThreadPool();
   private URL urlToProcess;

   public CompletableFuture<String> readPage(URL url)
   {
      return CompletableFuture.supplyAsync(() ->
         {
            try
            {
               var contents = new String(url.openStream().readAllBytes(),
                  StandardCharsets.UTF_8);
               System.out.println("Read page from " + url);
               return contents;
            }
            catch (IOException e)
            {
               throw new UncheckedIOException(e);
            }
         }, executor);
   }

   public List<URL> getImageURLs(String webpage) // nie czasochłonne
   {
      try
      {
         var result = new ArrayList<URL>();
         Matcher matcher = IMG_PATTERN.matcher(webpage);
         while (matcher.find())
         {
            var url = new URL(urlToProcess, matcher.group(1));
            result.add(url);
         }
         System.out.println("Znalezione adresy URL: " + result);
         return result;
      }
      catch (IOException e)
      {
         throw new UncheckedIOException(e);
      }
   }

   public CompletableFuture<List<BufferedImage>> getImages(List<URL> urls)
   {
      return CompletableFuture.supplyAsync(() ->
         {
            try
            {
               var result = new ArrayList<BufferedImage>();
               for (URL url : urls)
               {
                  result.add(ImageIO.read(url));
```

```
                  System.out.println("Załadowano " + url);
               }
               return result;
            }
            catch (IOException e)
            {
               throw new UncheckedIOException(e);
            }
         }, executor);
   }

   public void saveImages(List<BufferedImage> images)
   {
      System.out.println("Zapisywanie " + images.size() + " images");
      try
      {
         for (int i = 0; i < images.size(); i++)
         {
            String filename = "/tmp/image" + (i + 1) + ".png";
            ImageIO.write(images.get(i), "PNG", new File(filename));
         }
      }
      catch (IOException e)
      {
         throw new UncheckedIOException(e);
      }
      executor.shutdown();
   }

   public void run(URL url)
         throws IOException, InterruptedException
   {
      urlToProcess = url;
      CompletableFuture.completedFuture(url)
         .thenComposeAsync(this::readPage, executor)
         .thenApply(this::getImageURLs)
         .thenCompose(this::getImages)
         .thenAccept(this::saveImages);

      /*
      // ewentualnie można użyć eksperymentalnego klienta HTTP:

      HttpClient client = HttpClient.newBuilder().executor(executor).build();
      HttpRequest request = HttpRequest.newBuilder(urlToProcess.toURI()).GET()
         .build();
      client.sendAsync(request, BodyProcessor.asString())
         .thenApply(HttpResponse::body).thenApply(this::getImageURLs)
         .thenCompose(this::getImages).thenAccept(this::saveImages);
      */
   }

   public static void main(String[] args)
         throws IOException, InterruptedException
   {
      new CompletableFutureDemo().run(new URL("http://horstmann.com/index.html"));
   }
}
```

12.7.3. Czasochłonne zadania w wywołaniach zwrotnych interfejsu użytkownika

Jednym z powodów używania wątków jest możliwość usprawnienia interakcji z programem. Szczególnie ważne jest to w aplikacjach z interfejsem użytkownika. Kiedy program ma do wykonania jakieś czasochłonne zadanie, nie można go pozostawić w wątku interfejsu użytkownika, ponieważ spowoduje to jego zamrożenie. Zamiast tego należy uruchomić nowy wątek.

Jeśli na przykład trzeba odczytać plik, gdy użytkownik kliknie przycisk, nie należy tego robić w taki sposób:

```
var open = new JButton("Open");
open.addActionListener(event ->
    { // ŹLE — czasochłonna czynność wykonywana w wątku interfejsu
        var in = new Scanner(file);
        while (in.hasNextLine())
        {
            String line = in.nextLine();
            . . .
        }
    });
```

Zadanie to należy wykonać w osobnym wątku.

```
open.addActionListener(event ->
    { // DOBRZE — czasochłonne zadanie w osobnym wątku
        Runnable task = () ->
        {
            var in = new Scanner(file);
            while (in.hasNextLine())
            {
                String line = in.nextLine();
                . . .
            }
        };
    executor.execute(task);
});
```

Sęk w tym, że interfejsu użytkownika nie można zaktualizować bezpośrednio z wątku roboczego wykonującego czasochłonne zadanie. Interfejsy użytkownika, na przykład Swing, JavaFX czy Android, nie są bezpieczne wątkowo. Nie można odwoływać się do elementów interfejsu użytkownika z wielu wątków, ponieważ grozi to ich uszkodzeniem. JavaFX i Android pilnują tego i zgłaszają wyjątek, gdy programista próbuje uzyskać dostęp do interfejsu użytkownika z innego wątku niż wątek tego interfejsu.

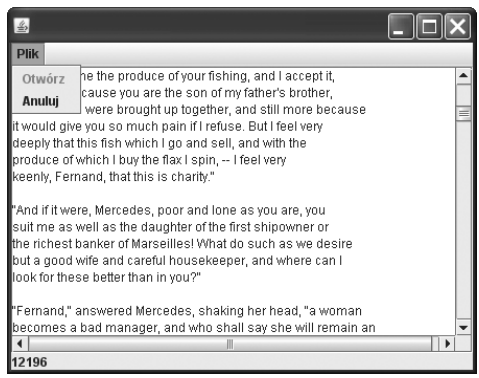
W związku z tym wszelkie zmiany w interfejsie użytkownika muszą być dokonywane w jego wątku. Każda biblioteka UI zawiera mechanizm umożliwiający zaplanowanie wykonania obiektu `Runnable` w wątku interfejsu użytkownika. Na przykład w bibliotece Swing stosuje się wywołanie:

```
EventQueue.invokeLater(() -> label.setText(percentage + "% complete"));
```

Implementacja informacji zwrotnej od użytkownika w wątku roboczym jest kłopotliwa, dlatego każda biblioteka interfejsu użytkownika zawiera jakąś klasę pomocniczą, na przykład `SwingWorker` w Swing, `Task` w JavaFX czy `AsyncTask` w Androidzie. Programista określa działania w ramach czasochłonnego zadania (które jest wykonywane w osobnym wątku), jak również aktualizacje postępu i ostateczną dyspozycję (które są wykonywane w wątku interfejsu użytkownika).

Program przedstawiony na listingu 12.11 posiada polecenia ładowania pliku tekstowego i anulowania tego procesu. Aplikację tę należy testować na pliku o dużych rozmiarach, jak *The Count of Monte Cristo* znajdującym się w katalogu *gutenberg* razem z katalogami z kodem. Plik jest ładowany w osobnym wątku. W trakcie tej operacji polecenie *Otwórz* w menu *Plik* jest nieaktywne, a *Anuluj* aktywne (rysunek 12.6). Po wczytaniu każdej linijki tekstu aktualizowany jest licznik w pasku stanu. Po ukończeniu ładowania polecenie *Otwórz* staje się z powrotem aktywne, polecenie *Anuluj* nieaktywne, a w pasku stanu wyświetla się napis *Zakończono*.

Rysunek 12.6. Ładowanie pliku w osobnym wątku



Listing 12.11. swingWorker/SwingWorkerTest.java

```
package swingWorker;

import java.awt.*;
import java.io.*;
import java.nio.charset.*;
import java.util.*;
import java.util.List;
import java.util.concurrent.*;

import javax.swing.*;

/**
 * Program demonstrujący wątek roboczy wykonujący potencjalnie czasochłonne zadanie.
 * @version 1.12 2018-03-17
 * @author Cay Horstmann
 */
public class SwingWorkerTest
{
   public static void main(String[] args) throws Exception
   {
      EventQueue.invokeLater(() -> {
         var frame = new SwingWorkerFrame();
```

```
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
         });
      }
}

/**
 * Ramka mająca obszar tekstowy pokazujący zawartość pliku tekstowego, menu pozwalające otworzyć plik
 * i anulować proces otwierania pliku oraz wiersz stanu pokazujący postęp ładowania pliku
 */
class SwingWorkerFrame extends JFrame
{
   private JFileChooser chooser;
   private JTextArea textArea;
   private JLabel statusLine;
   private JMenuItem openItem;
   private JMenuItem cancelItem;
   private SwingWorker<StringBuilder, ProgressData> textReader;
   public static final int TEXT_ROWS = 20;
   public static final int TEXT_COLUMNS = 60;

   public SwingWorkerFrame()
   {
      chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));

      textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
      add(new JScrollPane(textArea));

      statusLine = new JLabel(" ");
      add(statusLine, BorderLayout.SOUTH);

      var menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      var menu = new JMenu("Plik");
      menuBar.add(menu);

      openItem = new JMenuItem("Otwórz");
      menu.add(openItem);
      openItem.addActionListener(event -> {
         // Wyświetlenie okna dialogowego wyboru pliku
         int result = chooser.showOpenDialog(null);

         // Jeśli plik został wybrany, zostanie on ustawiony jako ikona etykiety
         if (result == JFileChooser.APPROVE_OPTION)
         {
            textArea.setText("");
            openItem.setEnabled(false);
            textReader = new TextReader(chooser.getSelectedFile());
            textReader.execute();
            cancelItem.setEnabled(true);
         }
      });

      cancelItem = new JMenuItem("Anuluj");
      menu.add(cancelItem);
      cancelItem.setEnabled(false);
```

```
      cancelItem.addActionListener(event -> textReader.cancel(true));
      pack();
   }

   private class ProgressData
   {
      public int number;
      public String line;
   }

   private class TextReader extends SwingWorker<StringBuilder, ProgressData>
   {
      private File file;
      private StringBuilder text = new StringBuilder();

      public TextReader(File file)
      {
         this.file = file;
      }

      // Poniższa metoda jest wykonywana w wątku roboczym - nie operuje na komponentach Swing

      public StringBuilder doInBackground() throws IOException, InterruptedException
      {
         int lineNumber = 0;
         try (var in = new Scanner(new FileInputStream(file), StandardCharsets.UTF_8))
         {
            while (in.hasNextLine())
            {
               String line = in.nextLine();
               lineNumber++;
               text.append(line).append("\n");
               var data = new ProgressData();
               data.number = lineNumber;
               data.line = line;
               publish(data);
               Thread.sleep(1); // Test operacji anulowania, nie ma potrzeby robienia
            }
         }
         return text;
      }

      // Poniższe metody są wykonywane w wątku dystrybucji zdarzeń

      public void process(List<ProgressData> data)
      {
         if (isCancelled()) return;
         var builder = new StringBuilder();
         statusLine.setText("" + data.get(data.size() - 1).number);
         for (ProgressData d : data) builder.append(d.line).append("\n");
         textArea.append(builder.toString());
      }

      public void done()
      {
         try
         {
            StringBuilder result = get();
```

```
                textArea.setText(result.toString());
                statusLine.setText("Done");
            }
            catch (InterruptedException ex)
            {
            }
            catch (CancellationException ex)
            {
                textArea.setText("");
                statusLine.setText("Anulowano");
            }
            catch (ExecutionException ex)
            {
                statusLine.setText("" + ex.getCause());
            }

            cancelItem.setEnabled(false);
            openItem.setEnabled(true);
         }
      };
   }
```

Program ten demonstruje typowy wygląd interfejsu użytkownika podczas wykonywania zadania w tle:

- Po każdym etapie pracy następuje aktualizacja interfejsu użytkownika w celu pokazania postępu.
- Po zakończeniu pracy w interfejsie dokonywana jest ostateczna zmiana.

Dzięki klasie `SwingWorker` zadanie to jest łatwe do wykonania. Wystarczy przedefiniować metodę `doInBackground`, aby wykonywała czasochłonne działania, i co jakiś czas wywoływać metodę `publish` mającą na celu pokazanie postępu. Metoda ta jest wykonywana w wątku roboczym. Metoda `publish` powoduje wykonanie metody `process` w wątku dystrybucji zdarzeń. Jej zadaniem jest obsługa danych dotyczących postępu. Po zakończeniu pracy w wątku dystrybucji zdarzeń wywoływana jest metoda `done` pozwalająca zakończyć aktualizację interfejsu użytkownika.

Aby wykonać jakieś działania w wątku roboczym, należy utworzyć obiekt klasy `SwingWorker` (każdy taki obiekt może być użyty tylko jeden raz). Następnie należy wywołać metodę `execute`. Metodę tę z reguły wywołuje się na rzecz wątku dystrybucji zdarzeń, ale nie jest to wymogiem.

Z założenia obiekt klasy `SwingWorker` powinien zwrócić jakiś wynik. Dlatego klasa `Swing`
`↪Worker<T, V>` implementuje interfejs `Future<T>`. Wynik ten można pobrać za pomocą metody `get` tego interfejsu. Ponieważ metoda ta włącza blokadę, dopóki wynik nie jest dostępny, nie należy wywoływać jej bezpośrednio po metodzie `execute`. Dobrym rozwiązaniem jest wywoływanie jej dopiero wówczas, gdy wiadomo, że praca została zakończona. Zazwyczaj metodę `get` wywołuje się w metodzie `done` (wywołanie metody `get` nie jest konieczne — czasami wystarczy przetworzenie danych postępu).

Zarówno pośrednie dane postępu, jak i końcowy wynik mogą być dowolnego typu. Typy te są określone w klasie `SwingWorker` jako parametry typowe. Klasa `SwingWorker<T, V>` tworzy wynik typu `T` i dane postępu typu `V`.

Do anulowania zadania w toku służy metoda `cancel` z interfejsu `Future`. Kiedy zadanie jest anulowane, metoda `get` zgłasza wyjątek `CancellationException`.

Jak już wiemy, wywołanie w wątku roboczym metody `publish` spowoduje wywołanie metody `process` na rzecz wątku dystrybucji zdarzeń. Aby zwiększyć wydajność, wyniki zwrócone przez kilka wywołań metody `publish` można zgrupować w jednym wywołaniu metody `process`. Metoda `process` odbiera obiekt `List<V>` zawierający wszystkie wyniki pośrednie.

Użyjemy tej techniki do wczytywania pliku tekstowego. Okazuje się, że komponent `JTextArea` jest niezbyt szybki. Dodawanie linii tekstu z dużego pliku tekstowego (jak *The Count of Monte Cristo*) zajmuje dużo czasu.

Aby pokazać użytkownikowi, że coś się dzieje, w pasku stanu będziemy wyświetlać liczbę wczytanych linijek tekstu. Dlatego dane postępu składają się z aktualnej liczby linii tekstu oraz aktualnej linii tekstu. Dane te pakujemy w prostej klasie wewnętrznej:

```
private class ProgressData
{
   public int number;
   public String line;
}
```

Ostateczny wynik stanowi tekst, który został wczytany do obiektu typu `StringBuilder`. W związku z tym klasa, której potrzebujemy, to `SwingWorker<StringBuilder, ProgressData)`.

Metoda `doInBackground` wczytuje dane z pliku wiersz po wierszu. Po każdym wierszu wywołujemy metodę `publish` publikującą numer i zawartość aktualnej linii.

```
@Override public StringBuilder doInBackground() throws IOException, InterruptedException
{
   int lineNumber = 0;
   var in = new Scanner(new FileInputStream(file), StandardCharsets.UTF_8);
   while (in.hasNextLine())
   {
      String line = in.nextLine();
      lineNumber++;
      text.append(line).append("\n");
      var data = new ProgressData();
      data.number = lineNumber;
      data.line = line;
      publish(data);
      Thread.sleep(1);  // Test operacji anulowania, nie ma potrzeby robienia tego w swoich programach.
   }
   return text;
}
```

Po każdej linii tekstu usypiamy wątek na jedną milisekundę, aby można było spokojnie przetestować anulowanie. Oczywiście w programach przeznaczonych do użytku nie należy tego robić, aby ich nie spowalniać. Jeśli postawimy przed tym wierszem symbol komentarza, zauważymy, że tekst książki wczytuje się dość szybko i jest tylko kilka większych aktualizacji interfejsu użytkownika.

Metoda `process` ignoruje wszystkie linie tekstu poza ostatnią oraz łączy wszystkie linie w jednej aktualizacji obszaru tekstowego.

```
@Override public void process(List<ProgressData> data)
{
   if (isCancelled()) return;
   var b = new StringBuilder();
   statusLine.setText("" + data.get(data.size() - 1).number);
   for (ProgressData d : data) b.append(d.line).append("\n");
   textArea.append(b.toString());
}
```

W metodzie `done` obszar tekstowy jest aktualizowany kompletnym tekstem, a polecenie *Anuluj* zostaje wyłączone.

Warto zwrócić uwagę na sposób uruchomienia obiektu klasy `SwingWorker` w nasłuchiwaczu zdarzeń elementu menu *Otwórz*.

Ta prosta technika pozwala na wykonywanie czasochłonnych zadań przy zachowaniu wrażliwości interfejsu użytkownika.

`javax.swing.SwingWorker<T, V>` **6**

- `abstract T doInBackground()`

  Tę metodę należy przedefiniować, aby wykonywała zadanie w tle i zwracała wynik swojego działania.

- `void process(List<V> data)`

  Tę metodę należy przedefiniować, aby przetwarzała pośrednie dane przetwarzania w wątku dystrybucji zdarzeń.

- `void publish(V... data)`

  Przesyła pośrednie dane postępu do wątku dystrybucji zdarzeń. Należy ją wywoływać w metodzie `doInBackground`.

- `void execute()`

  Planuje wykonanie obiektu klasy `SwingWorker` w wątku roboczym.

- `SwingWorker.StateValue getState()`

  Sprawdza stan obiektu `SwingWorker` — `PENDING`, `STARTED` lub `DONE`.

12.8. Procesy

Do tej pory pokazywaliśmy, jak wykonywać kod Javy w osobnych wątkach w obrębie jednego programu. Czasami jednak konieczne jest wykonanie innego programu. Do tego służą klasy `ProcessBuilder` i `Process`. Klasa `Process` wykonuje polecenie w osobnym procesie systemu operacyjnego oraz umożliwia komunikację poprzez standardowe strumienie wejścia, wyjścia i błędów. Klasa `ProcessBuilder` umożliwia konfigurację obiektu `Process`.

Klasa `ProcessBuilder` to elastyczniejsze zastępstwo dla wywołań *Runtime.exec*.

12.8.1. Budowanie procesu

Najpierw należy określić polecenie, które ma zostać wykonane. Można je przekazać w postaci obiektu `List<String>` lub po prostu łańcuchów tworzących całość polecenia.

```
var builder = new ProcessBuilder("gcc", "myapp.c");
```

Pierwszy łańcuch musi być poleceniem wykonywalnym, a nie wbudowanym poleceniem powłoki. Aby na przykład w systemie Windows wykonać polecenie `dir`, należy zbudować proces z łańcuchów `"cmd.exe"`, `"/C"` oraz `"dir"`.

Każdy proces ma katalog roboczy, na bazie którego są określane ścieżki względne katalogów podrzędnych. Domyślnie katalog roboczy procesu jest tożsamy z katalogiem maszyny wirtualnej, czyli najczęściej tym, w którym uruchomiono program. Można go zmienić za pomocą metody `directory`:

```
builder = builder.directory(path.toFile());
```

Każda z metod do konfiguracji obiektów `ProcessBuilder` zwraca ten obiekt, dzięki czemu można tworzyć łańcuchy poleceń. Ostatecznie zawsze wygląda to tak:

```
Process p = new ProcessBuilder(command).directory(file).…start();
```

Następnie należy określić, co ma się stać ze standardowymi strumieniami wejścia, wyjścia i błędów procesu. Domyślnie każdy z nich jest potokiem, do którego można uzyskać dostęp w następujący sposób:

```
OutputStream processIn = p.getOutputStream();
InputStream processOut = p.getInputStream();
InputStream processErr = p.getErrorStream();
```

Pamiętaj, że strumień wejściowy procesu jest strumieniem wyjściowym w JVM! Cokolwiek programista w nim zapisze, trafia na wejście procesu. I analogicznie, programista odczytuje to, co proces zapisze w strumieniach wyjściowym i błędów. Dla programisty są to strumienie wejściowe.

Programista może zdecydować, by strumienie wejściowy, wyjściowy i błędów nowego procesu były takie same jak strumienie JVM. Jeśli użytkownik uruchomi JVM w konsoli, wpisywane przez niego dane są przekazywane do procesu, którego wyniki także są pokazywane w konsoli. Aby zdefiniować to ustawienie dla wszystkich trzech strumieni, należy zastosować następujące wywołanie:

```
builder.redirectIO()
```

Jeśli potrzebne jest uzgodnienie tylko niektórych strumieni, należy przekazać wartość `Process ↪Builder.Redirect.INHERIT` do metody `redirectInput`, `redirectOutput` lub `redirectError`. Na przykład:

```
builder.redirectOutput(ProcessBuilder.Redirect.INHERIT);
```

Do plików strumienie procesów można przekierowywać przy użyciu obiektów `File`:

```
builder.redirectInput(inputFile)
    .redirectOutput(outputFile)
    .redirectError(errorFile)
```

Pliki strumieni wyjściowego i błędów są tworzone lub obcinane w chwili uruchomienia procesu. Aby zamiast tego dołączać dane na końcu, należy dodać poniższy wiersz kodu:

```
builder.redirectOutput(ProcessBuilder.Redirect.appendTo(outputFile));
```

Dobrym pomysłem jest połączenie strumieni wyjściowego i błędów, aby dane wyjściowe i komunikaty o błędach były pokazywane w kolejności ich generowania przez proces. W tym celu należy zastosować następujące wywołanie:

```
builder.redirectErrorStream(true)
```

Jednak od tej pory nie można już wywoływać metody `redirectError` obiektu `ProcessBuilder` lub `getErrorStream` obiektu `Process`.

Czasami konieczne jest modyfikowanie zmiennych środowiskowych procesu. W tym przypadku składnia budowy łańcuchowej zawodzi. Najpierw należy utworzyć środowisko budowniczego (które jest zainicjalizowane zmiennymi środowiskowymi procesu, w którym działa JVM), a następnie dodawać lub usuwać wpisy.

```
Map<String, String> env = builder.environment();
env.put("LANG", "fr_FR");
env.remove("JAVA_HOME");
Process p = builder.start();
```

Aby można było przekazać wynik jednego procesu na wejście innego (tak jak się to robi za pomocą operatora | w powłoce), w Javie 9 zdefiniowano metodę `startPipeline`. Należy przekazać listę budowniczych procesów oraz odczytać wynik ostatniego z nich. Poniżej znajduje się przykład programu wyszukującego rozszerzenia plików w drzewie katalogów:

```
List<Process> processes = ProcessBuilder.startPipeline(List.of(
new ProcessBuilder("find", "/opt/jdk-9"),
new ProcessBuilder("grep", "-o", "\\.[^./]*$"),
new ProcessBuilder("sort"),
new ProcessBuilder("uniq")
));
Process last = processes.get(processes.size() - 1);
var result = new String(last.getInputStream().readAllBytes());
```

Oczywiście w tym konkretnym przypadku efektywniejszym rozwiązaniem byłoby przejrzenie katalogów w Javie zamiast uruchamiania czterech procesów. Objaśnienie tej techniki znajduje się w rozdziale 2. tomu II.

12.8.2. Uruchamianie procesu

Po zakończeniu konfiguracji budowniczego można uruchomić proces, wywołując metodę `start`. Jeśli strumienie wejścia, wyjścia i błędów skonfigurowano jako strumienie, można teraz pisać w strumieniu wejściowym oraz odczytywać dane ze strumieni wyjścia i błędów. Na przykład:

```
Process process = new ProcessBuilder("/bin/ls", "-l")
    .directory(Path.of("/tmp").toFile())
    .start();
try (var in = new Scanner(process.getInputStream())) {
    while (in.hasNextLine())
        System.out.println(in.nextLine());
}
```

Strumienie procesów mają ograniczoną przestrzeń buforową. Nie należy zalewać wejścia oraz od razu powinno się odczytywać dane z wyjścia. Jeśli danych wejściowych i wyjściowych jest dużo, może być konieczne wytwarzanie i odbieranie ich w osobnych wątkach.

Aby poczekać na zakończenie pracy przez proces, należy wywołać:

```
int result = process.waitFor();
```

lub, aby nie czekać w nieskończoność:

```
long delay = . . .;
if (process.waitfor(delay, TimeUnit.SECONDS)) {
    int result = process.exitValue();
    . . .
} else {
    process.destroyForcibly();
}
```

Pierwsze wywołanie metody `waitFor` zwraca wartość wyjściową procesu (standardowo `0` oznacza powodzenie, a każda inna wartość — błąd). Drugie wywołanie zwraca `true`, jeśli proces nie przekroczy limitu czasu. Następnie należy pobrać wartość wyjściową za pomocą metody `exitValue`.

Zamiast czekać, aż proces się zakończy, można pozostawić go w działaniu i tylko od czasu do czasu sprawdzać, czy nadal działa, za pomocą metody `isAlive`. Aby zamknąć proces, należy wywołać metodę `destroy` lub `destroyForcibly`. Różnica między nimi zależy od platformy. W systemie UNIX pierwsza zamyka proces przy użyciu sygnału `SIGTERM` — a druga przy użyciu `SIGKILL`. (Metoda `supportsNormalTermination` zwraca `true`, jeśli metoda `destroy` może zamknąć proces normalnie).

Informację o zakończeniu działania procesu można też otrzymać asynchronicznie. Wywołanie `process.onExit()` zwraca obiekt `CompletableFuture<Process>`, przy użyciu którego można zaplanować wykonanie dowolnego działania.

```
process.onExit().thenAccept(p -> System.out.println("Wartość wyjścia: " +
p.exitValue()));
```

12.8.3. Uchwyty procesów

Aby zdobyć więcej informacji o procesie, który został uruchomiony przez program, lub o jakimkolwiek innym, który aktualnie działa w komputerze, należy użyć interfejsu `Process ↪Handle`. Obiekt `ProcessHandle` można utworzyć na cztery sposoby:

1. Dla danego obiektu `Process p`, `p.toHandle()` zwraca jego obiekt `ProcessHandle`.
2. Dla danego długiego identyfikatora procesu systemu operacyjnego `ProcessHandle.of(id)` zwraca uchwyt tego procesu.
3. `Process.current()` to uchwyt procesu, w którym działa ta maszyna wirtualna Javy.
4. `ProcessHandle.allProcesses()` zwraca obiekt `Stream<ProcessHandle>` zawierający wszystkie procesy systemu operacyjnego, które są widoczne dla bieżącego procesu.

Mając uchwyt procesu, można sprawdzić jego identyfikator, proces nadrzędny oraz procesy potomne:

```
long pid = handle.pid();
Optional<ProcessHandle> parent = handle.parent();
Stream<ProcessHandle> children = handle.children();
Stream<ProcessHandle> descendants = handle.descendants();
```

Egzemplarze typu `Stream<ProcessHandle>` zwracane przez metody `allProcesses`, `children` i `descendants` pokazują tylko migawkę stanu. Zanim programista przejrzy wyniki, każdy proces może zostać zamknięty oraz mogą zostać uruchomione inne.

Metoda `info` zwraca obiekt `ProcessHandle.Info` zawierający metody umożliwiające pobranie informacji o procesie.

```
Optional<String[]> arguments()
Optional<String> command()
Optional<String> commandLine()
Optional<String> startInstant()
Optional<String> totalCpuDuration()
Optional<String> user()
```

Wszystkie te metody zwracają wartości typu `Optional`, ponieważ nie każdy system operacyjny musi udostępniać wszystkie te informacje.

Do monitorowania i wymuszonego zamykania procesów interfejs `ProcessHandle` posiada te same metody (`isAlive`, `supportsNormalTermination`, `destroy`, `destroyForcibly` oraz `onExit`) co klasa `Process`. Nie ma natomiast odpowiednika metody `waitFor`.

`java.lang.ProcessBuilder` **5**

- `ProcessBuilder(String... command)`
- `ProcessBuilder(List<String> command)`

  Tworzy budowniczego procesu przy użyciu podanego polecenia i przekazanych argumentów.

- `ProcessBuilder directory(File directory)`

  Ustawia katalog roboczy procesu.

- `ProcessBuilder inheritIO()` **9**

  Zmusza proces do korzystania ze standardowych strumieni wejścia, wyjścia i błędów maszyny wirtualnej.

- `ProcessBuilder redirectErrorStream(boolean redirectErrorStream)`

  Jeśli parametr `redirectErrorStream` ma wartość `true`, standardowy strumień błędów procesu zostaje połączony ze standardowym strumieniem wyjściowym.

- `ProcessBuilder redirectInput(File file)` **7**
- `ProcessBuilder redirectOutput(File file)` **7**
- `ProcessBuilder redirectError(File file)` **7**

  Przekierowuje standardowe strumienie wejścia, wyjścia i błędów procesu do danego pliku.

- `ProcessBuilder redirectInput(ProcessBuilder.Redirect source)` **7**
- `ProcessBuilder redirectOutput(ProcessBuilder.Redirect destination)` **7**
- `ProcessBuilder redirectError(ProcessBuilder.Redirect destination)` **7**

  Przekierowuje standardowe strumienie wejścia, wyjścia i błędów procesu do `destination`. Parametr ten może mieć jedną z następujących wartości:

  - `Redirect.PIPE` — domyślne ustawienie, dostęp przez obiekt `Process`
  - `Redirect.INHERIT` — strumień z maszyny wirtualnej
  - `Redirect.DISCARD`
  - `Redirect.from(file)`
  - `Redirect.to(file)`
  - `Redirect.appendTo(file)`

- `Map<String,String> environment()`

  Zwraca słownik, który można modyfikować, do ustawiania zmiennych środowiskowych procesu.

- `Process start()`

  Uruchamia proces i zwraca jego obiekt `Process`.

- `static List<Process> startPipeline(List<ProcessBuilder> builders)` **9**

  Uruchamia potok procesów, łącząc standardowe wyjście każdego ze standardowym wejściem następnego.

`java.lang.Process 1.0`

- `abstract OutputStream getOutputStream()`

  Pobiera strumień umożliwiający zapis w strumieniu wejściowym procesu.

- `abstract InputStream getInputStream()`
- `abstract InputStream getErrorStream()`

  Tworzy strumień wejściowy umożliwiający odczyt ze strumienia wyjściowego lub błędów procesu.

- `abstract int waitFor()`

  Czeka, aż proces zakończy działanie, i zwraca wartość wyjścia.

- `boolean waitFor(long timeout, TimeUnit unit)` **8**

  Czeka na zakończenie procesu, ale nie dłużej niż określony czas. Zwraca `true`, jeśli proces zostanie zamknięty.

- `abstract int exitValue()`

  Zwraca wartość wyjścia procesu. Tradycyjnie wartość wyjściowa różna od zera oznacza błąd.

- `boolean isAlive()` **8**

  Sprawdza, czy proces jeszcze działa.

- `abstract void destroy()`

- `Process destroyForcibly()` **8**

  Zamyka ten proces normalnie lub wymusza zamknięcie.

- `boolean supportsNormalTermination()` **9**

  Sprawdza, czy dany proces można zamknąć normalnie, czy konieczne jest wymuszenie jego zniszczenia.

- `ProcessHandle toHandle()` **9**

  Zwraca obiekt `ProcessHandle` opisujący ten proces.

- `CompletableFuture<Process> onExit()` **9**

  Zwraca obiekt `CompletableFuture`, który zostaje wykonany po zamknięciu tego procesu.

`java.lang.ProcessHandle` **9**

- `static Optional<ProcessHandle> of(long pid)`
- `static Stream<ProcessHandle> allProcesses()`
- `static ProcessHandle current()`

  Zwraca uchwyty: procesu o określonym identyfikatorze, wszystkich procesów lub procesu maszyny wirtualnej.

- `Stream<ProcessHandle> children()`
- `Stream<ProcessHandle> descendants()`

  Zwraca uchwyty procesów potomnych tego procesu.

- `long pid()`

  Zwraca identyfikator tego procesu.

- `ProcessHandle.Info info()`

  Zwraca szczegółowe informacje o tym procesie.

`java.lang.ProcessHandle.Info` **9**

- `Optional<String[]> arguments()`
- `Optional<String> command()`
- `Optional<String> commandLine()`
- `Optional<Instant> startInstant()`
- `Optional<Instant> totalCpuDuration()`
- `Optional<String> user()`

  Zwraca wybraną informację, jeśli jest dostępna.

W tym miejscu kończy się pierwszy tom książki *Java. Podstawy*. Opisano w nim podstawy języka Java oraz niektóre fragmenty jego API, które są potrzebne w większości projektów programistycznych. Mamy nadzieję, że podobała Ci się podróż przez podstawowe zagadnienia związane z Javą i że udało Ci się tu znaleźć przydatne wiadomości. Dodatkowe informacje na temat systemu modułów Javy, funkcji sieciowych, zaawansowanych technik programowania interfejsu użytkownika i grafiki, bezpieczeństwa aplikacji czy internacjonalizacji zostały zawarte w drugim tomie.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Y

Z

JAVA — OTO JĘZYK MISTRZÓW PROGRAMOWANIA!

Świat usłyszał o Javie pod koniec 1995 roku. Wkrótce stała się niezwykle popularną i cenioną technologią. Dziś jest dojrzałym, rozbudowanym, elastycznym, a przy tym niezwykle starannie zaprojektowanym językiem programowania, który może służyć do pisania dużych systemów, małych programów, aplikacji mobilnych i aplikacji WWW. Charakteryzuje się też wysublimowanymi zabezpieczeniami, które w niego wbudowano. Każdy, kto chce pisać dobry i wydajny kod, powinien dobrze poznać zarówno podstawowe, jak i zaawansowane cechy Javy.

Ta książka jest kolejnym, zaktualizowanym i uzupełnionym wydaniem kultowego podręcznika dla profesjonalnych programistów Javy — to pierwszy tom, w którym omówiono podstawy języka oraz najważniejsze zagadnienia związane z programowaniem interfejsu użytkownika. W tym wydaniu opisano pakiet JDK Java Standard Edition (SE) w wersji 9, 10 i 11. Teorii towarzyszą liczne przykłady kodu, obrazujące zasady działania niemal każdej przedstawionej tu funkcji czy biblioteki. Przykładowe programy są proste, aby ułatwić naukę najważniejszych zagadnień.

W TEJ KSIĄŻCE MIĘDZY INNYMI:

- solidne wprowadzenie do Javy i przygotowanie środowiska pracy
- zasady programowania obiektowego: klasy, hermetyzacja, dziedziczenie
- mechanizm refleksji i obiekty proxy
- interfejsy, klasy wewnętrzne i wyrażenia lambda
- programowanie generyczne i system kolekcji
- GUI i praca z pakietem Swing
- programowanie współbieżne

CAY S. HORSTMANN — profesor informatyki na Uniwersytecie Stanowym w San Jose, autor popularnych podręczników do nauki Javy. Został wyróżniony tytułem Java Champion. Często zabiera głos na konferencjach programistycznych.

PRENTICE HALL
PEARSON EDUCATION

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-5778-5
9 788328 357785

Cena: 99,00 zł

ALWAYS LEARNING
PEARSON