Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

XIII. Java po C++. Programowanie wielowątkowe

1. Wprowadzenie

1.1. Program wielowątkowy zawiera dwie lub więcej działających współbieżnie ścieżek wykonywania. Każda z tych ścieżek nosi nazwę wątku. Java zawiera wbudowaną obsługę programowania wielowątkowego.

1.2. Pojęcie wątku wiąże się ściśle z pojęciem wielozadaniowości. Wielozadaniowość może bazować na procesach (współbieżne wykonywanie wielu programów) lub na wątkach (współbieżne wykonywanie zadań „pochodzących" od tego samego programu). Ta druga wielozadaniowość wprowadza mniejsze narzuty komunikacyjne: wątki, w odróżnieniu od procesów, współdzielą tę samą przestrzeń adresową.

1.3. Wszystkie klasy bibliotek Javy są pisane z myślą o wielowątkowości. Dzięki intensywnemu używaniu wątków, całe środowisko Javy nabiera charakteru asynchronicznego.

1.4. Alternatywą dla programowania wielowątkowego jest podejście zwane pętlą zdarzeń z odpytywaniem. W takim systemie jednowątkowym, gdy wątek blokuje się (zwykle – zawiesza swoje działanie w oczekiwaniu na pewien zasób), cały program przestaje działać.

2. Właściwości wątków

2.1. Wątek występuje w jednym z następujących stanów:
– działania (wykonywania),
– oczekiwania na wykonanie,
– zawieszenia (taki wątek można wznowić od miejsca uprzedniego zatrzymania),

– zablokowania (oczekiwania na zasoby),
– przerwania (przerwany wątek nie podlega wznowieniu).

2.2. Java każdemu z wątków przypisuje priorytet wskazujący, jak dany wątek powinien być traktowany względem pozostałych. Zmiana aktualnego wątku nazywa się przełączaniem kontekstu. Następuje ono w wypadku:
– dobrowolnego zrzeczenia się czasu procesora przez wątek (ustąpienie, uśpienie, zablokowanie wątku na operacji wejścia-wyjścia),
– wywłaszczenia przez wątek o wyższym priorytecie.

2.3. Konieczna w pewnych wypadkach synchronizacja działania wątków odbywa się w Javie poprzez tzw. monitory. Każdy obiekt posiada własny, niejawny monitor, który uruchamia się automatycznie w momencie wywołania jednej z synchronizowanych metod obiektu. Gdy jeden z wątków znajdzie się wewnątrz takiej metody, to pozostałe wątki nie mogą wywołać żadnej z synchronizowanych metod tego obiektu.

2.4. Wątek powstaje na bazie klasy stanowiącej rozszerzenie klasy standardowej `Thread` lub na bazie klasy implementującej interfejs `Runnable` (wyjątkiem – wątek główny).

3. Tworzenie wątków

3.1. Podczas uruchamiania programu w Javie powoływany jest automatycznie do życia tzw. wątek główny. Od niego pochodzą wszystkie wątki potomne i on też zazwyczaj kończy działanie całego programu. Wątkiem tym można sterować za pomocą obiektu klasy `Thread`, po wcześniejszym pobraniu do niego referencji przy użyciu metody `currentThread`, np.

```
class CurrentDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();
    System.out.println(”Aktualny watek: ” + t);
    t.setName(”Moj watek”);
    System.out.println(”Po zmianie nazwy: ” + t);
    try {
      for(int n=5; n>0; n--) {
      System.out.println(n);
      Thread.sleep(1000);
      }
    }
    catch (InterruptedException e) {
      System.out.println
                  (”Przerwanie watku glownego”);
    }
  }
}
```

Wynikiem działania tego programu będzie:

```
Aktualny wątek: Thread[main.5.main]
Po zmianie nazwy: Thread[Moj watek.5.main]
5
4
3
2
1
```

3.2. Najprostszy sposób utworzenia wątku potomnego to utworzenie klasy implementującej interfejs `Runnable`. Implementacja tego interfejsu wymaga jedynie zdefiniowania metody `run()`. Metoda `run()` ustanawia początek i koniec wykonywania nowego, współbieżnego wątku w programie. Oto przykład:

```
class NewThread implements Runnable {
  Thread t;
  NewThread() {
    t = new Thread(this, ”Przykładowy watek”);
    System.out.println(”Watek potomny: ” + t);
    t.start();
  }
  public void run() {
    try {
      for (int i=5; i>0; i--) {
        System.out.println
          (”Watek potomny: ” + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println
          (”Wyjście z wątku potomka.”);
      }
  }
}

class ThreadDemo {
  public static void main(String args[]) {
    new NewThread();
    try {
      for(int i=5; i>0; i--) {
        System.out.println(”Watek glowny: ” + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println
       (”Przerwano dzialanie glownego watku.”);
      }
    System.out.println(”Wyjscie z watku glownego.”);
  }
}
```

3.3. Drugi sposób polega na zdefiniowaniu klasy rozszerzającej klasę `Thread`, a następnie utworzeniu obiektu tej klasy. Nowa klasa musi koniecznie zawierać metodę przesłaniającą metodę `run()`, stanowiącą początek kodu wątku. W celu zrealizowania działań objętych poprzednim przykładem, (rozszerzającą) klasę `NewThread` należałoby zdefiniować jak następuje:

```
class NewThread extends Thread {
    NewThread() { super(”Przykladowy watek”);
    System.out.println
      (”Watek potomny: ” + this);
    start();
  }
// reszta jak wyzej
}
```

Rezultat działania obu przytoczonych programów będzie identyczny:

```
Watek potomny: Thread[Przykladowy watek.5.main]
Watek głowny: 5
Watek potomny: 5
Watek potomny: 4
Watek głowny: 4
Watek potomny: 3
Watek potomny: 2
Watek głowny: 3
Watek potomny: 1
Watek głowny: 2
Watek głowny: 1
Wyjscie z watku glownego
```

3.4. W sposób analogiczny do tworzenia dwóch wątków (głównego i jednego potomnego), można w programie utworzyć dowolną liczbę wątków.

4. Klasa `Thread` i jej metody

4.1. W klasie `Thread` można odnaleźć wiele metod istotnych z punktu widzenia synchronizowania działań wątków współbieżnych. W szczególności, są w niej metody umożliwiające uwarunkowanie zakończenia działania jednego wątku od zakończenia działania innych wątków:

– metoda `final boolean isAlive()` zwraca wartość `true`, gdy wątek dla którego została wywołana jest nadal wykonywany i wartość `false` w przeciwnym wypadku,
– metoda `final void join() throwsInterruptedException` wywołana dla określonego wątku powoduje, że wątek wywołujący czeka na zakończenie wskazanego.

4.2. Do ustalenia priorytetu wątku służy metoda o sygnaturze `final void setPriority(int` *poziom*`)`, zaś do pobrania aktualnego priorytetu – metoda `final int getPriority()`. Argument *poziom* metody `setPriority` oznacza nową wartość priorytetu, która musi być liczbą naturalną z przedziału `MIN_PRIORITY` do `MAX_PRIORITY`. Przywrócenie wątkowi priorytetu domyślnego odbywa się poprzez przekazanie wartości stałej `NORM_PRIORITY`, równej liczbie `5`. Wszystkie wymienione stałe są zdefiniowane w klasie `Thread` jako zmienne `static final`.
Korzystając z priorytetów, należy jednak w kwestii harmonogramowania wątków polegać mniej na wywłaszczaniu, niż na samoczynnym oddawaniu sterowania.

5. Synchronizacja dostępu do zasobów

5.1. W wypadku, gdy dwa lub więcej wątków korzysta ze wspólnego zasobu, należy zapewnić, by w każdym momencie tylko jeden z nich miał fizyczny dostęp do zasobu. Mechanizm takiego udostępniania nosi nazwę synchronizacji. Podstawowym narzędziem synchronizacji jest monitor, zwany też semaforem. Każdy obiekt klasy `Object` i dowolnej z podklas tej klasy posiada monitor, który zakłada blokadę wzajemnie wykluczającą. Gdy pewien wątek wejdzie do monitora,

pozostałe oczekują na jego wyjście. Wątek, który właśnie opuścił monitor, może wejść do niego ponownie.
Jeden z wbudowanych sposobów synchronizacji polega na definiowaniu metod z użyciem modyfikatora `synchronized`. Gdy wątek wywoła taką metodę, to wszystkie inne wątki, które będą próbowały się dostać do niej lub do innej synchronizowanej metody przedmiotowego obiektu, będą musiały poczekać na opuszczenie monitora przez wątek pierwszy, np.

```
class Callme {
  synchronized void call(String msg) {
    System.out.print(”[” + msg);
    try {
      Thread.sleep(1000);
    } catch(InterruptedException e) {
      System.out.println(”Przerwano”);
    }
    System.out.println(”]”);
  }
}

class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
  public Caller(Callme targ, String s) {
    target=targ;
    msg=s;
    t=new Thread(this);
    t.start();
  }
  public void run() {
    target.call(msg);
  }
}
```

```
class Synch {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, ”Witaj”);
    Caller ob2 = new Caller(target, ”Synchroniczny”);
    Caller ob3 = new Caller(target, ”Swiecie”);
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e)
      System.out.println(”Przerwano”);
    }
  }
}
```

```
[Witaj]
[Synchroniczny]
[Swiecie]
```

```
class Synch {
  public static void main(String args[]) {
    Callme target1 = new Callme();
    Callme target2 = new Callme();
    Callme target3 = new Callme();
    Caller ob1 = new Caller(target1,”Witaj”);
    Caller ob2 = new Caller(target2,”Synchroniczny”);
    Caller ob3 = new Caller(target3,”Swiecie”);
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e)
      System.out.println(”Przerwano”);
    }
  }
}
```

```
[Witaj[Synchroniczny[Swiecie]
]
]
```

Jaka sytuacja może spowodować uzyskanie efektu w postaci:

```
[Witaj[Swiecie]
[Synchroniczny]
]
?
```

```
class Synch {
  public static void main(String args[]) {
    Callme target1 = new Callme();
     Callme target2 = new Callme();
    Caller ob1 = new Caller(target1,”Witaj”);
    Caller ob2 = new Caller(target1,”Synchroniczny”);
    Caller ob3 = new Caller(target2,”Swiecie”);
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e)
      System.out.println(”Przerwano”);
    }
  }
}
```

5.2. Druga z metod synchronizacji dostępu polega na tworzeniu tzw. bloków synchronizowanych. Blok taki przyjmuje następującą, ogólną postać:

```
synchronized(obiekt) {
//instrukcje synchronizowane
}
```

Wejście do tego bloku przez pewien wątek jest równoznaczne z zajęciem przez ten wątek monitora obiektu wskazanego referencją *obiekt*. W monitorze może się równolegle znajdować tylko jeden blok synchronizowany lub jedna metoda synchronizowana obiektu.

```
class Table{
  void printTable(int n){
    synchronized(this) { //synchronized block
      for(int i=1;i<=5;i++){
        System.out.println(n*i);
        try {
          Thread.sleep(400);
        } catch(Exception e){System.out.println(e);}
      }
    }
  } //end of the method
}
class MyThread1 extends Thread{
  Table t;
  MyThread1(Table t) {
    this.t=t;
  }
  public void run() {
    t.printTable(5);
  }
}
class MyThread2 extends Thread{
  Table t;
  MyThread2(Table t) {
    this.t=t;
  }
  public void run() {
    t.printTable(100);
  }
}

public class TestSynchronizedBlock1 {
  public static void main(String args[]) {
    Table obj = new Table();//only one object
    MyThread1 t1=new MyThread1(obj);
    MyThread2 t2=new MyThread2(obj);
    t1.start();
    t2.start();
  }
}
```

5.3. Przy tworzeniu programów wielowątkowych należy szczególnie troszczyć się o to, by nie doprowadzić do tzw. blokady wzajemnej. Z taką sytuacją będziemy mieć do czynienia, gdy dwa wątki zaprogramujemy jako cyklicznie zależne od pary synchronizowanych obiektów.

5.4. Wyrafinowany poziom sterowania dostępem do zasobu można osiągnąć przy użyciu metod tzw. komunikacji międzyprocesowej. Są to zadeklarowane w klasie `Object` metody:

- `wait()`, powodująca oddanie przez wątek monitora i jego zapadnięcie w sen do momentu, dopóki inny wątek nie wejdzie do monitora i nie wywoła metody `notify()`;
- `notify()`, budząca wątek, który wywołał metodę `wait()` dla tego samego obiektu oraz
- `notifyAll()`, budząca wszystkie wątki, które wywołały metodę `wait()` dla tego samego obiektu. Spośród wybudzonych wątków tylko jeden uzyska dostęp do monitora.

```
class Q {
  int n;
  boolean valueSet = false;
  synchronized int get() {
    if(!valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println
          (”Zlapano InterruptedException”);
        }
    System.out.println(”Pobrano: ” + n);
    valueSet=false;
    notify();
    return n;
  }
  synchronized void put(int n) {
    if(valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println
          (”Zlapano InterruptedException”);
        }
    this.n= n;
    valueSet= true;
    System.out.println(”Wlozono: ” + n);
    notify();
  }
}
```

```
class Producer implements Runnable {
  Q q;
  Producer(Q q) {
    this.q=q;
    new Thread(this, ”Producent”).start();
  }
  public void run() {
    int i=0;
    while(true) {
      q.put(i++);
    }
  }
}

class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
    this.q=q;
    new Thread(this, ”Konsument”).start();
  }
  public void run() {
    while(true) {
      q.get();
    }
  }
}
class PCFixed {
  public static void main(String args[]) {
  Q q = new Q();
  new Producer(q);
  new Consumer(q);
  System.out.println(”Zakończ.”);
  }
}
```

Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

6. Zawieszanie, wznawianie i zatrzymywanie wątków

6.1. W starszych wersjach Javy procesami tymi zarządzały metody `suspend()`, `resume()` oraz `stop()`, zdefiniowane w klasie `Thread`. Metody te wycofano w wersji Java 2, ze względu na liczne, poważne błędy systemowe.

6.2. Obecnie, do sterowania wykonywaniem wątku wykorzystuje się metody `wait()` i `notify()`, odziedziczone po klasie `Object`.

Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

Przykład ze strony: http://www.tutorialspoint.com/java/java_thread_control.htm

```
class RunnableDemo implements Runnable {
   public Thread t;
   private String threadName;
   boolean suspended = false;
   RunnableDemo( String name){
       threadName = name;
       System.out.println("Creating " +  threadName );
   }
   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 10; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", "
                                                          + i);
            // Let the thread sleep for a while.
            Thread.sleep(300);
            synchronized(this) {
            while(suspended) {
               wait();
            }
          }
         }
     } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + "
                                             interrupted.");
     }
     System.out.println("Thread " +  threadName + "
                                                exiting.");
   }
   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
   void suspend() {
      suspended = true;
   }
   synchronized void resume() {
      suspended = false;
       notify();
   }
}
public class TestThread {
   public static void main(String args[]) {
```

```
        RunnableDemo R1 = new RunnableDemo( "Thread-1");
        R1.start();

        RunnableDemo R2 = new RunnableDemo( "Thread-2");
        R2.start();

        try {
            Thread.sleep(1000);
            R1.suspend();
            System.out.println("Suspending First Thread");
            Thread.sleep(1000);
            R1.resume();
            System.out.println("Resuming First Thread");
            R2.suspend();
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);
            R2.resume();
            System.out.println("Resuming thread Two");
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        try {
            System.out.println("Waiting for threads to finish.");
            R1.t.join();
            R2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

Wynik działania:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
```

```
Running Thread-1
Thread: Thread-1, 10
Running Thread-2
Thread: Thread-2, 10
Thread: Thread-1, 9
Thread: Thread-2, 9
Thread: Thread-1, 8
Thread: Thread-2, 8
Thread: Thread-1, 7
Thread: Thread-2, 7
Suspending First Thread
Thread: Thread-2, 6
Thread: Thread-2, 5
Thread: Thread-2, 4
Resuming First Thread
Suspending thread Two
Thread: Thread-1, 6
Thread: Thread-1, 5
Thread: Thread-1, 4
Thread: Thread-1, 3
Resuming thread Two
Thread: Thread-2, 3
Waiting for threads to finish.
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
Main thread exiting.
```