Metody programowania

Wprowadzenie do platformy Java

Dr inż. Andrzej Grosser

Częstochowa, 2013

Spis treści

1. Wielowątkowość **5**

1.1. Wątki 5

1.1.1. Stany wątków 6

1.1.2. Właściwości wątków 6

1.1.3. Synchronizacja wątków 7

1.2. Wielowątkowość a Swing 8

Literatura **11**

1. Wielowątkowość

Wykonywanie wielu zadań można przyśpieszyć stosując rozdzielając je na wiele podzadań, które mogą być wykonywane równolegle. W programowaniu korzysta się wtedy z mechanizmów wielowątkowości. Zagadnienie to jest jednak złożone i rozległe, rozdział przybliża jedynie podstawowe zagadnienia. Do napisania rozdziału posłużyła książka "Core Java"[2] i "Thinking in Java"[1] i dokumentacji Javy.

1.1. Wątki

Współczesne programowanie kładzi duży nacisk na programowanie równoległe i współbieżne. Wynika to z tego, że obecnie używane systemy operacyjne wspierają wielozadaniowość a procesory posiadają wiele rdzeni, które można wykorzystać do wykonywania wielu obliczeń równolegle.

W kontekście wielowątkowości pojawią się pojęcia procesu i wątku. Rozróżnia się je ze względu na sposób przechowywania danych. Proces posiada swoje dane, natomiast wątki współdzielą wspólne dane. Poza tym utworzenie wątku wiąże się zazwyczaj z dużo mniejszym nakładem prac ze strony systemu operacyjnego niż w przypadku procesów. Ponadto komunikacja pomiędzy wątkami jest bardziej efektywna niż pomiędzy procesami.

Wątki są implementowane w Javie za pomocą klasy `Thread`. Uruchamianie nowego wątku składa się z następujących kroków:

1. Dostarczenie metody `run()` w klasie implementującej interfejs `Runnable` (może być w klasie anonimowej, której obiekt będzie przekazywany bezpośrednio, jako argument konstruktora wątku):

```
1 class MojUruchamiacz implements Runnable {
2     public void run() {
3     }
4 }
```

2. Utworzenie obiektu tej klasy:

```
1 Runnable r = new MojUruchamiacz();
```

3. Utworzenie instancji klasy `Thread` z wykorzystaniem wcześniejszego obiektu:

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

4. Wystartowanie wątku:

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

1.1.1. Stany wątków

Wątki mogą znajdować w jednym z sześciu stanów:

- nowy - znajduje się w nim wątek po utworzeniu za pomocą operatora new, wątek jeszcze nie jest uruchomiony,
- wykonywalny - wątek znajduje się w tym stanie po wywołaniu metody start(), nie musi być w tym stanie od razu uruchamiany, gdyż przydział zasobów zależy od systemu operacyjnego.
- zablokowany - wątek jest czasowo nieaktywny, jest to spowodowane oczekiwanie na przydział zasobu, do którego ma dostęp w tym momencie inny wątek,
- oczekujący - podobnie jak we wcześniej opisanym stanie wątek jest czasowy nieaktywny, jest to związane z oczekiwaniem wątku na uzyskanie powiadomienia z innego wątku o osiągnięciu planowanego warunku,
- czasowo oczekujący - wątek oczekuje na upłynięcie określonego odcinka czasu (np. po wywołaniu metody sleep()).
- zakończony - wątek przechodzi w ten stan, gdy jego metoda run() zakończyła normalnie lub poprzez nieobsłużony wyjątek.

1.1.2. Właściwości wątków

Priorytet wątków Każdy wątek posiada swój własny priorytet. Planista wątku preferuje wątki o wyższym priorytecie. Priorytet ustawia się metodą `setPriority()`. Wartość priorytetu może być zapisana pomiędzy stałymi `MIN_PRIORITY` (wartość 1) i `MAX_PRIORITY` (wartość 10). Stała `NORM_PRIORITY` ma wartość 5.

Wątki uruchomione jako demony Wątki demony są to wątki, których zadaniem jest tylko obsługa innych - na przykład wątki, które powiadamiają inne wątki o upłynięciu określonego okresu czasu. Wątek staje się demonem poprzez wywołanie:

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

Uchwyty dla nieprzechwyconych wyjątków Metoda run() nie może wywoływać żadnego sprawdzalnego wyjątku, ale może być zakończone przez nieobsłużony wyjątek. W tej sytuacji nie ma jednak żadnej klauzuli catch, do której wyjątek mógłby być przekazany. Zamiast tego, po tym jak wątek zostanie zakończony, wyjątek jest przekazywany do uchwytu wyjątku.

Uchwyt nieprzechwyconego wyjątku musi należeć do klasy implementującej interfejs `Thread.UncaughtExceptionHandler`. Wspomniany interfejs dostarcza metodę o sygnaturze `void uncaughtException(Thread t, Throwable e)`.

Uchwyt instaluje się w wątku za pomocą metody `setUncaughtExceptionHandler()`.

1.1.3. Synchronizacja wątków

Często w aplikacjach korzystających z wielu wątków, dwa lub więcej z nich musi korzystać z dostępu do wspólnych danych. W takiej sytuacji, może się zdarzyć, że jeden z wątków może nadpisać pobrane wcześniej przez inny wątek dane, przez co uzyskuje się niepoprawne wyniki działania programu. Dla przykładu jeden z wątku programu księgowego pobrał najpierw stan konta, żeby dodać do niego wpłaconą gotówkę, w tym samym czasie inny wątek zaksięgował wypłatę. Pierwszy wątek zakończył swoje działanie później, i napisał stan konta z uwzględnieniem jedynie wpłaty, wypłata nie została ostatecznie zaksięgowana, gdyż operacja ta została wykonana pomiędzy pobraniem stanu konta a modyfikacją stanu konta przez pierwszy wątek. Z inną sytuacją można byłoby się spotkać, gdy to pierwszy wątek zakończył swoje operacje, przed zakończeniem drugiego wątku, wówczas to wpłata nie zostałaby uwzględniona w stanie konta. Taka przypadek jest nazywany sytuacją wyścigu(ang. *race condition*). Fragment kodu, w którym używa się współdzielonego zasobu jest nazywany sekcją krytyczną.

Są dwa sposoby ochrony bloku kodu przed współbieżnym dostępem - słowo kluczowe `synchronized`:

```
1 class X {
2     //...
3     public synchronized void f() {
4                 //...
5             }
6 }
```

i klasa `ReentrantLock`:

```
class X {
  private final ReentrantLock lock = new ReentrantLock();
  //...

  public void f() {
    lock.lock();

    try {
      // ...
    }
    finally {
      lock.unlock()
    }
  }
}
```

Dla metod oznaczonych słowem kluczowym synchronized nie jest możliwe ich dwukrotne wywołanie w tym samym czasie na rzecz innych obiektów (innymi słowy wywołania nie mogą się przeplatać). Jeżeli wątek wykonuje metodę synchronizowaną, to inne wątki, które mogłyby chcieć ją wywołać, czekają aż pierwszy dokończy pracę.

W Javie 1.5 wprowadzono klasę ReentrantLock z pakietu java.util.concurrent. Mechanizm blokowania zastosowany w tej klasie wymaga wywołania metody `lock()` w celu ochrony danej sekcji kodu przed dostępem do niej wielu wątków a następnie odblokowania dostępu metodą `unlock()`. Najlepszym miejscem wywołania drugiej z wymienionych metod jest sekcja finally, umożliwia to odblokowanie kodu nawet w sytuacji wystąpienia nieobsłużonego wyjątku.

1.2. Wielowątkowość a Swing

Użycie wątków przy budowie graficznego programu może spowodować, że będzie on lepiej reagował na operacje użytkownika. Dla przykładu dobrym pomysłem jest umieszczenie w osobnym wątku operacji, które zajmują dużo czasu. Dzięki temu interfejs użytkownika nie będzie blokowany na czas wykonywania tych operacji. Należy też przy tym pamiętać, że operacje na komponentach Swing muszą wykonywane jedynie w wątku obsługi komunikatów. Biblioteka Swing nie gwarantuje bezpieczeństwa operacji na komponentach przeprowadzonych z różnych wątków.

Poniższy kod pokazuje różną obsługę tej samej operacji:

```
1  public class MainFrame extends JFrame {
2      public MainFrame() {
3          setControls();
4          setSize(640, 480);
5      }
6
7      private void setControls() {
8          //...
9          JButton button1 = new JButton("Przycisk 1");
10         button1.addActionListener(new ActionListener() {
11
12             @Override
13             public void actionPerformed(ActionEvent e) {
14                 btn1ActionPerformed(e);
15             }
16         });
17         JButton button2 = new JButton("Przycisk 2");
18         button2.addActionListener(new ActionListener() {
19
20             @Override
21             public void actionPerformed(ActionEvent e) {
22                 btn2ActionPerformed(e);
23             }
24         });
25         //...
26     }
27
28     private void btn1ActionPerformed(ActionEvent evt) {
29         for(int i = 0; i < 1000000; ++i)
30             System.out.println(i);
31     }
32
33     private void btn2ActionPerformed(ActionEvent evt) {
34         Thread t = new Thread(new Runnable() {
35             public void run() {
36                 for(int i = 0; i < 1000000; ++i)
37                     System.out.println(i);
38             }
39         });
40         t.start();
41     }
42
43
```

```
44      public static void main(String[] args) {
45          EventQueue.invokeLater(new Runnable() {
46              public void run() {
47                  MainFrame frame = new MainFrame();
48                  frame.setDefaultCloseOperation(JFrame.
                        EXIT_ON_CLOSE);
49                  frame.setVisible(true);
50              }
51          });
52      }
53  }
```

Obsługa operacji wymagającej dużej ilości czasu (jest ona symulowana poprzez wypisywanie w konsoli miliona liczb) jest realizowana w podanym kodzie na dwa sposoby:

1. za pomocą przycisku ”Metoda 1” - operacja jest od razu wykonywana w bieżącym wątku,
2. za pomocą przycisku ”Metoda 2” - operacja jest oddelegowana do nowego wątku.

Można zaobserwować, że w pierwszym przypadku interfejs użytkownika zastyga na czas wykonywania operacji (np. nie da się przesunąć pasków bocznych przy komponencie listy, zwinięcie i rozwinięcie okna nie powoduje jego odświeżenia), natomiast w drugim przypadku interfejs reaguje na operacje użytkownika pod wypisywania kolejnych liczb w konsoli.

Literatura

[1] B. Eckel. *Thinking in Java*. Helion, 2006.

[2] C. S. Horstmann and G. Cornell. *Core Java Volume I–Fundamentals*. Prentice Hall, 2011.