Programowanie współbieżne i rozproszone

Tomasz Olas

Katedra Informatyki
Politechnika Częstochowska

Wątki w Javie

Współbieżność w Javie

- Współbieżność w Javie jest realizowana w oparciu o model wielowątkowy.
- Wielowątkowość w Javie opiera się na konstrukcjach obiektowych (klasy, interfejsy, metody).
- Jest ona wbudowana w podstawy języka:
  - klasa bazowa **Object** w Javie zawiera metody związane z przetwarzaniem współbieżnym - ***wait()***, ***notify()***, ***notifyAll()***.
  - każdy obiekt może stać się elementem przetwarzania współbieżnego.
- Korzystanie ze wspólnych obiektów i ich atrybutów przez wiele wątków prowadzi do tych samych problemów, które występowały przy realizacji współbieżności w językach proceduralnych.

Wątek główny

- Za uruchamianie wątków i zarządzanie nimi odpowiada klasa **Thread** (pakiet ***java.lang***).
- Każdy program ma przynajmniej jeden wykonujący się wątek (wątek główny).
- Tworzony jest automatycznie i w Javie wykonuje on ciąg instrukcji umieszczonych w metodzie ***main()***.
- Watek główny powstaje jako pierwszy i może utworzyć dodatkowe wątki.
- W Javie jest możliwe uzyskanie odniesienia do bieżącego wątku (w tym watku głównego) poprzez wywołanie statycznej metody klasy **Thread**:

```
public static Thread currentThread()
```

Klasa **Thread** - wybrane metody

- Uruchamianie i zatrzymywanie wątków:
  - ***start*** - uruchomienie wątku polegające na wykonaniu metody ***run***,
  - ***stop*** – zatrzymanie wątku (niezalecany sposób zatrzymania wątku),
  - ***run*** - metoda wykonywana przez wątek,
  - ***sleep*** - zawieszenie działania wątku na określony czas.
- Identyfikacja wątków:
  - ***currentThread*** - zwraca identyfikator wątku,
  - ***getName*** - odczytanie nazwy wątku,
  - ***setName*** - umożliwia ustawienie nazwy wątku,
  - ***isAlive*** - sprawdzenie czy wątek działa,
  - ***toString*** - uzyskanie atrybutów wątku.
- Priorytety wątków:
  - ***setPriority*** - ustawienie priorytetu wątku,
  - ***getPriority*** - odczytanie priorytetu wątku.

Zawieszenie wątku

- Wątek może zostać zawieszony („uśpiony”) na określony czas (w milisekundach) przy użyciu metody ***sleep*** z klasy **Thread**:

```
public static native void sleep(long millis) throws
                                                InterruptedException
```

- Przykład - zawieszenie działania wątku głównego na 1 sekundę:

```
class MyClass {
    public static void main(String args[]) {
        System.out.println("Wątek główny: START");
        try {
                Thread.sleep(1000);
        }
        catch(InterruptedException e) {}
        System.out.println("Wątek główny: KONIEC");
    }
}
```

Sposoby tworzenia wątków

- Aby utworzyć nowy wątek należy:
  - utworzyć obiekt klasy, która zawiera metodę przeznaczoną do rozpoczęcia przetwarzania współbieżnego - ***run***,
  - uruchomić wątek (obiekt klasy **Thread**), który będzie wykonywał tą metodę.
- Klasę nowego wątku można utworzyć na trzy sposoby:
  - zdefiniowanie klasy dziedziczącej po klasie **Thread**,
  - zdefiniowanie klasy implementującej interfejs **Runnable**,
  - zdefiniowanie klasy implementującej interfejs **Callable**.
- Drugi i trzeci sposób jest stosowany, gdy klasa wątku musi dziedziczyć po innej niż **Thread** klasie (Java nie wspiera mechanizmu wielodziedziczenia klas, a zamiast tego możliwe jest wielodziedziczenie interfejsów).

Tworzenie wątku - dziedziczenie po klasie **Thread**

- Aby utworzyć nowy wątek przy użyciu dziedziczenia po klasie **Thread** należy:

1. Utworzyć nową klasę jako potomną klasy **Thread**, która będzie zawierała metodę ***run()***. W metodzie tej należy umieścić kod, który będzie wykonywany w ramach tego wątku:

```
class MyThread extends Thread {
  public void run() {
    // kod wykonywany przez wątek
  }
}
Aby
```

2. Utworzyć obiekt tej klasy:

```
MyThread thread = new MyThread(...);
```

3. Uruchomić wątek poprzez wywołanie metody ***start()***:

```
thread.start();
```

Dziedziczenie po klasie **Thread** - przykład

- Kod programu (***MyThread.java***):

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Jestem nowym wątkiem");

    }
    public static void main(String args[]) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

- Kompilacja i uruchomienie:

```
> javac MyThread.java
> java -cp . MyThread
Jestem nowym wątkiem
```

Tworzenie wątku - implementacja interfejsu **Runnable**

1. Utworzyć nową klasę implementującej interfejs **Runnable** (w tym przypadku klasa może dziedziczyć po innej dowolnej klasie). Dodać do niej metodę ***run()***:

```
class MyClass extends OtherClass implements Runnable {
  public void run() {
    // kod wykonywany przez wątek
  }
}
```

2. Utworzyć obiekt tej klasy:

```
MyClass object = new MyClass(...);
```

3. Utworzyć obiekt klasy **Thread** przekazując jako parametr konstruktora referencję do utworzonego wcześniej obiektu klasy:

```
Thread thread = new Thread(object);
```

4. Uruchomić wątek poprzez wywołanie metody ***start()***:

```
thread.start();
```

Implementacja interfejsu **Runnable** - przykład

```
class MyClass implements Runnable {
    public void run() {
        try {
            for (int i = 0; i < 5; ++i) {
                Thread.sleep(500);
                System.out.println("Nowy wątek: " + i);
            }
        }
        catch(InterruptedException e) {}
    }
    public static void main(String args[]) {
        System.out.println("Wątek główny: START");
        MyClass object = new MyClass();
        Thread thread = new Thread(object);
        thread.start();
        try {
            for (int i = 0; i < 5; ++i) {
                Thread.sleep(500);
                System.out.println("Główny wątek: " + i);
            }
        }
        catch(InterruptedException e) {}
        System.out.println("Wątek główny: KONIEC");
    }
}
```

```
> javac MyClass.java
> java -cp . MyClass
Wątek główny: START
Nowy wątek: 0
Główny wątek: 0
Nowy wątek: 1
Główny wątek: 1
Nowy wątek: 2
Główny wątek: 2
Nowy wątek: 3
Główny wątek: 3
Nowy wątek: 4
Główny wątek: 4
Wątek główny: KONIEC
```

Zakończenie i przerwanie działania wątku

- Wątek, w naturalny sposób, kończy pracę, gdy zakończy się jego metoda ***run***.
- Wątek może zostać zakończony poprzez wywołanie metody ***stop***, ale się tego nie zaleca, ponieważ może to prowadzić do wejścia obiektów, z których korzysta w nieprawidłowy stan.
- Do przerwania działania wątku służy metoda ***interrupt*** (dokładnie wątek nie jest natychmiast przerwany, ale ustawiony zostaje sygnalizator przerwania). Przerwanie następuje jeżeli wątek znajdzie się wewnątrz metody dającej się przerwać, np. ***sleep***, ***join***, ***wait*** lub wywołana zostanie metoda ***interrupted***.
- ***interrupted*** - pozwala na sprawdzenie stanu sygnalizatora przerwania, po którym powinna nastąpić reakcja wątku na próbę przerwania.

Czekanie na zakończenie wątku

- Wywołanie metody ***join*** (klasa **Thread**) powoduje, że wątek wywołujący czeka na zakończenie działania wątku, na rzecz którego została ona wywołana:

```
public final void join();
```

- Przykład:

```
class MyThread extends Thread {
    public void run() {
        try {
                Thread.sleep(3000);
        } catch(InterruptedException e) {}
    }
    public static void main(String args[]) {
        System.out.println("Wątek główny: START");
        MyThread thread = new MyThread();
        thread.start();
        try {
                thread.join(); // wątek główny czeka na wątek thread
        } catch(InterruptedException e) {}
        System.out.println("Wątek główny: KONIEC");
    }
}
```

Stany wątku

- W trakcie działania wątki mogą znajdować się w jednym z czterech stanach:
  - Utworzony (*New Thread*) - obiekt wątku został utworzony za pomocą operatora ***new***, ale nie została jeszcze wykonana metoda start().
  - Wykonywalny (*Runnable*) - stan po wywołaniu metody ***start***, oznaczająca gotowość do wykonania, jak tylko procedura szeregująca przydzieli mu rdzeń lub procesor.
  - Wykonywany (*Running*) - wątek jest wykonywany na przydzielonym mu rdzeniu lub procesorze.
  - Nie wykonywany (*Not Runnable*) - wątek nie może być wykonywany z powodu braku pewnych zasobów, oczekuje na wykonanie jakiejś operacji np. synchronizujących lub został przeniesiony w ten stan za pomocą metod: ***suspend***, ***sleep***, ***wait***.
  - Zakończony (*Dead*) - wątek zakończony na skutek zakończenia metody ***run*** lub po wykonaniu metody ***stop*** (niezalecane).

Synchronizacja wątków

- Synchronizacja wątków w Javie jest oparta na mechanizmie zamków monitorowych (monitorów) - (*monitor lock*).
- Każdy obiekt w Javie zawiera monitor, którego zadaniem jest synchronizacja wątków w dostępie do obiektu - w danej chwili tylko jeden wątek może działać na tym obiekcie.
- Synchronizacja obiektów (sekcja krytyczna związana z obiektem) jest realizowana poprzez słowo kluczowe ***synchronized***.
- Synchronizacja może być realizowana na poziomie bloków (fragmentów kodu), bądź metod (nie mogą być synchronizowane konstruktory).

Sposoby synchronizacji wątków

- Synchronizacja na poziomie metod - słowo kluczowe ***synchronized*** występuje przy definiowaniu metody:

```
public synchronized void method() {
  ...
}
```

- Synchronizacja na poziomie bloków instrukcji (blok synchronizowany) - słowo kluczowe ***synchronized*** występuje przed blokiem instrukcji z argumentem będącym referencją do obiektu, w oparciu o który będzie realizowana synchronizacja (którego monitor ma zostać zajęty):

```
synchronized (object) {
  ...
}
```

Synchronizacja wątków - monitory

- Kiedy wątek wywołuje synchronizowaną metodę obiektu, to automatycznie zajmuje monitor tego obiektu.
- Inny wątek, wywołujący synchronizowaną metodę tego obiektu (niekoniecznie tą samą) zostanie zablokowany i będzie czekał na zakończenie wykonywania metody przez wątek, który zajął monitor tego obiektu.
- Po zakończeniu metody synchronizowanej monitor jest zwalniany i oczekujący wątek może uzyskać dostęp do obiektu.
- Monitor związany z danym obiektem można również zająć poprzez wykonanie bloku synchronizowanego powiązanego z danym obiektem.
- Z obiektem jest związany jeden monitor i nie ma znaczenia, w jaki sposób zostanie on zajęty - czy poprzez metodę synchronizowaną, czy przez blok synchronizowany (oba sposoby mogą być ze sobą łączone w jednym programie).

Synchronizowane metody - przykład

- Przykład klasy, która zawiera dwie synchronizowane metody operujące na atrybutach obiektu:

```
public class SynchronizowanaKlasa {

  private double a;
  private double b;

  public synchronized void dodajDoA(double value) {
    a += value;
  }

  public synchronized void dodajDoB(double value) {
    b += value;
  }
}
```

- Wadą tego rozwiązania w tym przypadku jest wykonywanie synchronizacji niezależnie, czy wątki operują na atrybucie ***a***, czy ***b***.

Synchronizacja wątków - monitory

- Często stosowanym rozwiązaniem w przypadku bloków synchronizowanych jest powiązanie bloku z obiektem, w którym ten blok się znajduje (poprzez referencję ***this***) - dzięki temu tylko część kodu metody jest synchronizowana.

```
public class SynchronizowanaKlasa {

  public void obliczenia() {
    ....                    // fragment nie wymagający synchronizacji
    synchronized (this) {
        ...                 // fragmaent wymagający synchronizacji
    }
  }
}
```

- Można również tworzyć obiekty, które pełnią tylko rolę pomocniczą w synchronizacji nie mając innych funkcji w programie.

Synchronizowane metody - przykład

- Przykład zastosowania dodatkowych obiektów do rozdzielenia synchronizacji w klasie, w zależności od wykorzystywanych w poszczególnych metodach atrybutów klasy:

```
public class SynchronizowanaKlasa {

  private double a;
  private Object mutexA = new Object();
  private double b;
  private Object mutexB = new Object();

  public void dodajDoA(double value) {
    synchronized (mutexA) {
      a += value;
    }
  }

  public void dodajDoB(double value) {
    synchronized (mutexB) {
      b += value;
    }
  }
}
```

Mechanizm powiadomień

- Do koordynacji pracy pomiędzy wątkami można wykorzystać mechanizm powiadomień.
- Z każdym obiektem w języku Java może być powiązany zbiór czekających wątków (*waiting set*).
- Mechanizm ten umożliwia zawieszenie i zwolnienie wątku do momentu, w którym zostanie spełniony określony warunek.
- Warunek ten może być dowolny i jest niezależny od samego obiektu, np. osiągnięcie przez zmienną określonej wartości.
- Do obsługi mechanizmu powiadomień służą metody z klasy **Object** - ***wait***, ***notify***, ***notifyAll***.
- Muszą one być wywoływane wewnątrz metody synchronizowanej lub w bloku synchronizowanym, aby uniknąć jednoczesnej próby oczekiwania i wysyłania powiadomienia.

Metoda *wait*

- Wywołanie metody ***wait*** na danym obiekcie powoduje, że wątek, który ją wykonał zostanie zawieszony do czasu, aż inny wątek nie wykona metody ***notify*** lub ***notifyAll*** dla tego samego obiektu.

```
public final void wait() throws InterruptedException;
```

- Wątek wywołujący zostanie dodany do zbioru oczekujących wątków obiektu.
- Wywołanie ***wait*** powoduje zwolnienie monitora, co umożliwia innemu wątkowi wysłanie powiadomienia:

```
synchronized (obj) {
  try {
    obj.wait();
  } catch (InterruptedException ie) { ... }
}
```

Metoda *notify*

- Do odblokowania jednego z czekających wątków (które wywołały metodę ***wait***) w mechanizmie powiadomień służy metoda ***notify***:

```
public final void notify();
```

- Nie jest określone, który z czekających wątków zostanie wznowiony.
- Odblokowany wątek nie zostanie natychmiast wykonany, ale musi zaczekać na zwolnienie monitora przez wątek zgłaszający powiadomienie.
- Ponadto będzie on konkurował z innymi wątkami o dostęp do synchronizowanego obiektu.
- Przykład:

```
synchronized (obj) {
  obj.notify();
}
```

Metoda *notifyAll*

- Odblokowanie wszystkich czekających wątków realizowane jest za pomocą metody ***notifyAll***:

```
public final void notifyAll();
```

- Podobnie, jak w przypadku metody ***notify***, odblokowane wątki nie zostaną natychmiast wykonane, ale muszą zaczekać na zwolnienie monitora przez wątek zgłaszający powiadomienie.
- Ponadto będą one konkurowały o dostęp do synchronizowanego obiektu.
- Przykład:

```
synchronized (obj) {
  obj.notifyAll();
}
```