Języki i paradygmaty programowania: Laboratorium nr 13

Podstawowe paradygmaty programowania obiektowego - wprowadzenie. Java - programowanie wielowątkowe.

2017-2018

mgr inż. Przemysław Walkowiak
dr inż. Michał Ciesielczyk

Instrukcja

W czasie pisania programu pamiętaj o:

1. dbaniu o czytelność kodu (odpowiednie formatowanie kodu, nazewnictwo zmiennych adekwatne do ich znaczenia, komentarze),
2. dbaniu o czytelność interfejsu z użytkownikiem (w sposób jawny pytaj użytkownika jakie dane ma podać oraz opisuj wyniki, które zwracasz),
3. przed fragmentem implementującym poszczególne zadania umieść komentarz: `/*Zadanie X */` oraz wypisz na ekranie analogiczny komunikat (X jest numerem zadania),
4. każde zadanie umieść w oddzielnej klasie z odpowiednimi metodami,
5. zaimplementuj menu wyboru zadania, a następnie wykorzystując pętle **do–while** oraz konstrukcję **switch** wykonaj odpowiedni fragment kodu,
6. w zadaniach wymagających udzielenia komentarza bądź odpowiedzi, należy umieścić go w kodzie programu (np. w postaci komentarza albo wydrukować na ekranie),
7. w zadaniach polegających na zaprojektowaniu klasy należy utworzyć jej instancję i wykorzystać zaimplementowaną funkcjonalność.

Wprowadzenie

W Javie, każdy wątek jest powiązany z instancją klasy `Thread`. Aplikacja tworząca instancję klasy `Thread` musi dostarczyć kod, który powinien zostać uruchomiony w danym wątku.

Najprostszym sposobem na zdefiniowanie operacji, które mają zostać wykonane przez dany wątek, jest przekazanie do konstruktora `Thread` obiektu implementującego interfejs `Runnable`.

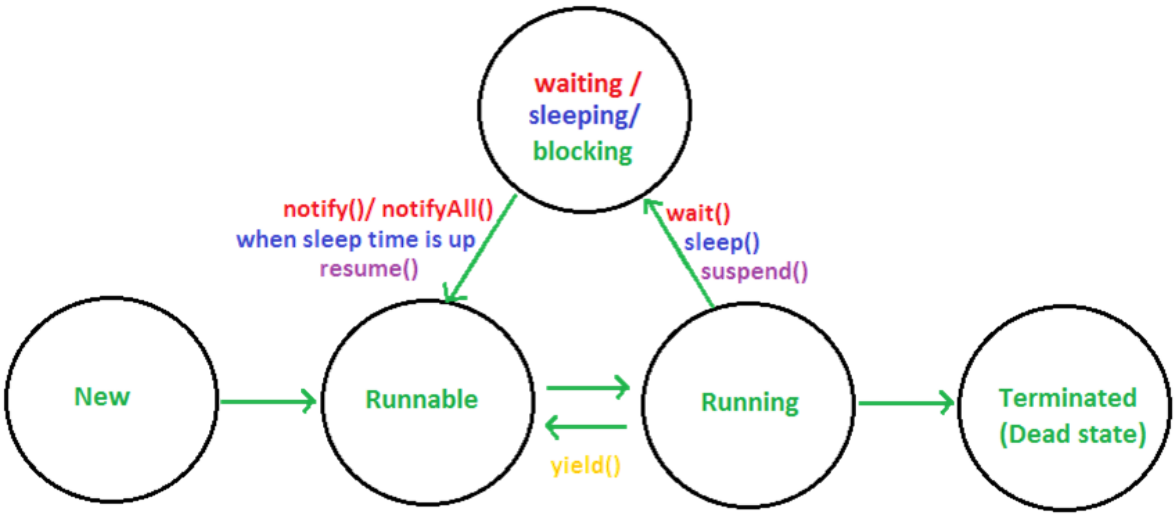
Alternatywnie, dla uproszczenia, można wykorzystać wyrażenia lambda np.:

```
(new Thread(() -> System.out.println("Hello from a thread!"))).start();
```

Na rysunku 1 przedstawiono poszczególne stany w jakich może się znaleźć wątek.

Dodatkowe informacje:

- Essential Java Classes – Concurency



Rysunek 1: Cykl życia wątku.

Zadania

Zadanie 1

Zdefiniuj klasę `Countdown` implementującą interfejs `Runnable` oraz posiadającą dwa pola prywatne (inicjalizowane w konstruktorze):

- `String name` – nazwę obiektu, oraz
- `int limit` – maksymalna wartość licznika.

W metodzie `Countdown.run()` umieść następujący fragment kodu:

```
    for (int i = limit; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
    }
```

Po przygotowaniu klasy `Countdown` w funkcji głównej uruchom nowy wątek w następujący sposób:

```
public static void main(String[] args) {
    Thread t1 = new Thread(new Countdown("t1", 10));
    t1.start();
    System.out.println("Main thread run is over");
}
```

Co zaobserwowałeś?

Spróbuj zatrzymać działanie wątku głównego do momentu zakończenia działania wątku `t1` w następujący sposób:

a) w pętli, sprawdzając czy wątek jest wciąż aktywny z wykorzystaniem metody `isAlive()`,
b) z wykorzystaniem metody `join()`.

Zastanów się, który ze sposobów jest bardziej wydajny?

Wskazówka 1 Pamiętaj o konieczności obsłużenia zgłaszanych wyjątków (tutaj: `InterruptedException`), np. poprzez wypisywanie odpowiedniego komunikatu.

Zadanie 2

Zmodyfikuj klasę `Countdown` w taki sposób by czas uśpienia w metodzie `Countdown.run()` był równy liczbie losowej z przedziału [10..1000] zamiast stałej wartości 1000. Następnie uruchom poniższy fragment kodu:

```
    Thread racer1 = new Thread(new Countdown("racer 1", 20));
    Thread racer2 = new Thread(new Countdown("racer 2", 20));

    racer1.start();
    racer2.start();

    racer1.join();
    racer2.join();

    System.out.println("Race is over");
```

Uruchom program kilkukrotnie. Czy wynik działania jest ten sam z każdym uruchomieniem? Czy jesteś w stanie przewidzieć, który wątek zakończy się jako pierwszy?

Zadanie 3

Zapoznaj się z implementacją klasy `Counter`:

```
public class Counter {

    protected long count = 0;

    public void add(long value) {
        this.count = this.count + value;
    }

    public long getCount() {
        return count;
    }
}
```

Wykorzystując powyższą implementację, utwórz w swoim programie zmienną `counter`:

```
Counter counter = new Counter();
```

Następnie, uruchom jednocześnie poniższy fragment kodu w trzech wątkach:

```
    for (int i=0;i<1000;i++){
        counter.add(1);
    }
```

Poczekaj aż każdy z wątków zakończy swoje działanie, a następnie wyświetl zawartość licznika (metoda `getCount()`). W komentarzu napisz dlaczego wynik działania programu jest różny od programu, w którym ten sam kod zostałby uruchomiony w jednym wątku.

Korzystając z modyfikatora **synchronized**, zmodyfikuj implementację klasy `Counter` zabezpieczając odpowiednio metodę `Counter.add()` przed dostępem z wielu wątków.

Zadanie 4

Utwórz kolekcję `numbers` typu `ArrayList` do przechowywania liczb całkowitych, a następnie kolejno:

a) utwórz t wątków (obiektów typu `Thread`), z których każdy powinien dodać do kolekcji `numbers` n liczb (losowo lub np. liczby od 1 do n)
b) uruchom wszystkie wątki,
c) zaczekaj aż wszystkie wątki zakończą swoje działanie.

Na koniec wypisz na ekran liczbę elementów kolekcji `numbers` (powinna być równa $t \times n$).

Uruchom swój program dla $t \in \{1, 2, 4\}$ oraz $n \in \{100, 1000, 10000\}$. Jak myślisz, dlaczego wynik jego działania nie zawsze będą poprawne?

Zmień typ kolekcji `numbers` na `Vector` i ponownie przetestuj swój program.

Jak myślisz, dlaczego po zmianie typu kolekcji program zawsze zwraca poprawne wyniki? Podpowiedź możesz znaleźć w dokumentacji lub implementacji klasy `Vector`.

Zadanie 5*

Zmodyfikuj program z zadania 4 z poprzednich zajęć w taki sposób aby strumień był przetwarzany wielowątkowo.

Wskazówka Skorzystaj z metody `BaseStream.parallel`.

Dodatkowe informacje:

- The Java Tutorials: Parallelism

Zadanie 6

Poniższy program ilustruje klasyczną interakcję między dwoma wątkami:

- Konsumentem (klasa `Consumer` przedstawiona na listingu 1), oraz
- Wytwórcą (klasa `Producer` przedstawiona na listingu 2).

Wątek wytwórcy tworzy wiadomości i umieszcza je w kolejce, podczas gdy konsument je czyta i wyświetla. Zakładamy, że wątek konsumenta „biegnie" dużo wolniej niż wątek wytwórcy. Oznacza to, że Wytwórca czasami musi zatrzymywać się i czekać, aż Konsument go „dogoni".

Listing 1: Consumer.java

```
package pl.poznan.put.lab12;

public class Consumer implements Runnable {

    private static final long DEFAULT_DELAY = 50;
    private final Producer producer;
    private final String name;
    private int messageCount = 0;

    public Consumer(String nazwa, Producer producent) {
        this.producer = producent;
        this.name = nazwa;
    }

    @Override
    public void run() {
        while (true) {
            try {
                String msg = producer.getMessage();
                if (msg != null) {
                    System.out.println(name + " received a message: " + msg);
                    messageCount++;
                } else {
                    System.err.println("Failed to receive a message!");
                }
                Thread.sleep(DEFAULT_DELAY);
            } catch (InterruptedException e) {
                System.out.println(name + " interrupted!");
                return;
            }
        }
    }

    public int getMessageCount() { return messageCount; }
}
```

Listing 2: Producer.java

```
package pl.poznan.put.lab12;

import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.LinkedList;
import java.util.Queue;

public class Producer implements Runnable {

    private static final int MAX_QUEUE = 5;
    private static final long DEFAULT_DELAY = 10;
    private final SecureRandom random = new SecureRandom();
    private final Queue<String> messages = new LinkedList<>();
    private int messageCount = 0;

    @Override
    public void run() {
        while (true) {
            try {
                putMessage();
                messageCount++;
                Thread.sleep(DEFAULT_DELAY);
            } catch (InterruptedException e) {
                System.out.println("Producer interrupted!");
                return;
            }
        }
    }

    public int getMessageCount() { return messageCount; }

    private void putMessage() throws InterruptedException {
        // FIXME: non thread-safe method
        while (messages.size() >= MAX_QUEUE) {
            Thread.sleep(100);
        }
        messages.add(new BigInteger(130, random).toString(32));
    }

    public String getMessage() throws InterruptedException {
        // FIXME: non thread-safe method
        while (messages.isEmpty()) {
            Thread.sleep(100);
        }
        return messages.poll();
    }
}
```

Przeanalizuj zamieszczoną implementację, oraz wskaż potencjalne problemy związane z klasą producenta – czy gwarantuje ona bezpieczeństwo wątków?

Uruchom poniższy kod (skorzystaj z załączonych plików). Zastanów się co powoduje błędy w otrzymywanych wynikach.

```
package pl.poznan.put.lab12;

public class Zadanie4 {

    public static void main(String args[]) throws InterruptedException {
        Producer producer = new Producer();
        Consumer consumer1 = new Consumer("Consumer 1 ", producer);
        Consumer consumer2 = new Consumer("Consumer 2 ", producer);

        Thread p = new Thread(producer);
        Thread c1 = new Thread(consumer1);
        Thread c2 = new Thread(consumer2);

        p.start();
        c1.start();
        c2.start();

        Thread.sleep(3000);
        p.interrupt();
        p.join();
        System.out.println("\nProduced " + producer.getMessageCount()
                + " messages.");

        Thread.sleep(500);
        c1.interrupt();
        c2.interrupt();
        c1.join();
        c2.join();
        System.out.println("\nConsumed " + (consumer1.getMessageCount()
                + consumer2.getMessageCount()) + " messages.");

        System.out.println("\nMain thread stopped.");
    }
}
```

Popraw implementację klasy `Producer` zmieniając metody `putMessage` oraz `getMessage` w taki sposób by były bezpieczne-wątkowo (*thread-safe*).

Wskazówka Skorzystaj z metod wait() oraz notify(). Więcej informacji: `https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html`