22. - Vlákna, Paralelní programování, Asynchroní metody, Concurrent patterns

Co je to vlákno?

Vlákno je v Javě jedna instance třídy Thread. Třída Thread umožňuje vykonával kód paralerně – to znamená, že dvě části programu mohou běžet najednou. Kdykoliv spustíme nějaké nové vlákno, JVM pro něj vytvoří vlastní samostatný zásobník na Stacku. Díky tomu může celé vlákno pracovat bezpečně s vlastními daty, aniž by nedocházelo k přepisování dat nějakého jiného vlákna.

Vlákna se JVM snaží různě dělit i mezi procesory, ale není to tak, že čtyři vlákna budou 100% na čtyřech procesorech. Je to ale hodně pravděpodobné. JVM multithreading umí.

Vlákna se v Javě dají implementovat dvěma způsoby.

Využití extendu

Jako první můžeme třídu podědit od třídy Thread. Poté přepíšeme a overridneme metodu `run`. Při spuštění se tedy vytvoří vlastní vlákno, na kterém budou probíhat věci, které jsou napsané v `run` metodě

```
public class Main extends Thread {
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Implementováním

Jako druhé můžeme třídu nechat implementovat Runnable. Poté přepíšeme a overridneme metodu `run`. Při spuštění se tedy vytvoří vlastní vlákno, na kterém budou probíhat věci, které jsou napsané v `run` metod. Také můžete vidět, že můžeme dát díky Runnable na vstup Threadu v Mainu třídu Task.

```
class Task implements Runnable {
    private String taskName;

    public Task(String name) {
        this.taskName = name;
    }

    public void run() {
        System.out.println("Task " + taskName + " is running in thread " + Thread.currentThread().getName());
    }
}
```

```
public class Main {
    public static void main(String[] args) {

        Thread thread1 = new Thread(new Task("Task 1"));
        Thread thread2 = new Thread(new Task("Task 2"));

        thread1.start();
        thread2.start();
    }
}
```

Paralelní programování

Paralelní programování je způsob, jak psát programy paralelně. Je opakem sekvenčního, které instrukce zpracovává postupně jednu po druhé. Paralelizace se využívá v případě, že máme nějaký algoritums na řešení šíleného a náročného problému. My jej rozdělíme na několik různých částí, které každé budou najeno počítač a pak společně problém vyřeší rychleji. Díky tomu nějaké části kódu nemusejí čekat na něco, co jim stejně nepředá data.

Jelikož se ovšem vlákna spouštějí najednou, vede to někdy v problémům, kdy program vrátí pokaždé jiný výsledek. Takovým příkladem může být takovýto kód:

```
public class Main extends Thread {
  public int amount = 0;

  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    System.out.println(thread.currentThread().getName());
    System.out.println(thread.amount);
    thread.amount++;
    System.out.println(thread.amount);
  }

  public void run() {
    System.out.println(Thread.currentThread().getName());
    amount++;
  }
}
```

V tomto kusu kódu se totiž vytvoří druhé vlákno od Mainu `(Main thread = new Main())`

První vlákno je samotný main. Toto druhé vlákno spustíme (pomocí `.start()` metody, což zavolá metodu `run()` a díky tomu se začnou tyto instrukce provádět na novém vlákně.

Výstupy se budou lišit, protože je proměnná amount sdílená mezi vlákny!

A výstupy… jsou opravdu různé.

```
Vlákno v mainu:main
Vlákno v runu: Thread-0
0
2
```

```
Vlákno v mainu:main
0
1
Vlákno v runu: Thread-0
```

```
Vlákno v mainu:main
Vlákno v runu: Thread-0
1
2
```

```
Vlákno v mainu:main
0
Vlákno v runu: Thread-0
1
```

Asynchronní metody

Asynchronní metody jsou takové metody, které neblokují svým spuštěním chod aplikace. Příkladem může být třeba spojení s databázovým serverem – Pokud máme slabé internetové spojení a bude komunikace a spojení trvat dlouho, třeba tři vteřiny, asynchronní metoda se bude s databázovým serverem spojovat tři vteřiny, ale program nebude tři vteřiny čekat, ale rovnou bude pokračovat v běhu a načítání různých dalších komponent.

Jakmile je asynchronní metoda dokončena, program pokračuje od toho řádku, kde byl využit join(). Cokoliv před Joinem tedy pokračuje a jede. Příklad:

```
import java.util.concurrent.CompletableFuture;

public class Main {
    public static void main(String[] args) {
        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
            System.out.println("Začínám asynchronní metodu!");
            try {
                Thread.sleep(3000);
                System.out.println("Právě jsem dokončil asynchronní metodu! ");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        System.out.println("Metoda main pokračuje ve svém běhu. Tento kód je před joinem");
        future.join();
        System.out.println("Tento kód je po joinu");
    }
}
```

V tomto příkladě my spustíme asynchronní metodu. Metoda main ale bude pokračovat dál a dál, dokud nebude `future.join().` Ten čeká na asynchronní metodu a dokud není hotová, tak on dál nebude pokračovat. Jakmile se asynchronní metoda dokončí, kód bude pokračovat.

Výsledek by mohl btý takový:

```
Metoda main pokračuje ve svém běhu. Tento kód je před joinem
Začínám asynchronní metodu!
Právě jsem dokončil asynchronní metodu!
Tento kód je po joinu
```

Ještě si můžete říct: Proč nejdříve main pokračuje a teprve poté se začíná s asynchronní metodou? Je to proto, že knihovna CompeatableFuture hodí asynchronní metodu do jiného vlákna. **Pozor, asynchronní metody nemusí vždy být v jiném vlákně!**

Úžasná věcička, že?

Concurrent design patterns

Jedná se o vzory, které řeší problémy, při kterých program běží na několika vláknech najednou a řeší souběžně určité operace. Mezi nejznámější patří například

- Thread Pool – Bazének několika vláken, které řeší určité problémy seřazené ve frontě.
- Read Write Lock – Tzv. RWL slouží k umožnění vláknům k přístupu pro čtení v jeden a ten samý čas.

Thread Pool

Thread pool je bazének vláken, které řeší různé problémy seřazené ve frontě a hlavní výhodou tohoto návrhového vzoru je dosažení konkurence – Stavu, ve kterém se výpočty a části programů mohou řešit mimo normální stanovený průběh.

Read Write Lock

Read Write Lock, také zvaný RWL umožňuje přístup k jednotlivým proměnným nebo polím v jeden moment. Write je exkluzivní, musí se použít lock. Jedná se také o synchronizační primitivum.

Zámky fungují na tom principu, že jedno vlákno přistoupí k metodě. Ta na prvním řádku spustí lock -> V tu chvíli do ní nemůže jiné vláknou vstoupit a musí počkat, až první vlákno dojede. Teprve poté se zámek odemkne, druhé vlákno může vstoupit do metody.

Locky zabraňují možnostem v jednom momentu zapsat na stejnou pozici item, přistoupit k něčemu co by už nemuselo existovat a podobné.