Vícevláknové aplikace

Jiří Vokřínek

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 8

B6B36PJV – Programování v JAVA

Využití vláken v GUI

Vlákna v GUI (Swing)

Rozšíření výpočetního modulu v aplikaci DemoBarComp o vlákno

Návrhový vzor Observer

Využití třídy SwingWorker

Diskutovaná témata

Část I

Využití vláken v GUI

Vlákna v GUI (Swing)

- Vlákna můžeme použít v libovolné aplikaci a tedy i v aplikaci s GUI.
- Vykreslování komponent Swing se děje v samostatném vlákně vytvořeném při inicializaci toolkitu
- Proto je vhodné aktualizaci rozhraní realizovat notifikací tohoto vlákna z jiného

  Snažíme se pokud možno vyhnout asynchronnímu překreslování z více vláken – race condition
- Zároveň se snažíme oddělit grafickou část od výpočetní (datové) části aplikace (MVC)

`http://docs.oracle.com/javase/tutorial/uiswing/concurrency`

Samostatné výpočetní vlákno pro výpočetní model v aplikaci DemoBarComp

- Třídu `Model` rozšíříme o rozhraní **Runnable**
- Vytvoříme novou třídu `ThreadModel`
  - Voláním metody `compute` spustíme samostatné vlákno
  - Musíme zabránit opakovanému vytváření vlákna

    Příznak `computing`
  - Metodu uděláme synchronizovanou
- Po stisku tlačítka stop ukončíme vlákno

  Implementujeme třídu `StopListener`
- Ve třídě `ThreadModel` implementuje metodu `stopComputation`

  Nastaví příznak ukončení výpočetní smyčky `end`

`lec07/DemoBarComp-simplethread`

Po spuštění výpočtu je GUI aktivní, ale neaktualizuje se *progress bar*, je nutné vytvořit vazbu s výpočetního vlákna – použijeme návrhový vzor **Observer**

Návrhový vzor **Observer**

- Realizuje abstraktní vazbu mezi objektem a množinou pozorovatelů
- Pozorovatel je předplatitel (*subscriber*) změn objektu
- Předplatitelé se musejí registrovat k pozorovanému objektu
- Objekt pak informuje (notifikuje) pozorovatele o změnách
- V Javě je řešen dvojicí třídy **Observable** a **Observer**

Výpočetní model jako **Observable** objekt 1/4

- **Observable** je abstraktní třídy
- `ThreadModel` již dědí od `Model`, proto vytvoříme nový **Observable** objekt jako instanci privátní třídy `UpdateNotificator`
- Objekt `UpdateNotificator` použijeme k notifikaci registrovaných pozorovatelů

```
public class ThreadModel extends Model implements
    Runnable {
   private class UpdateNotificator extends Observable {
      private void update() {
         setChanged();       // force subject change
         notifyObservers(); // notify reg. observers
      }
   }
   UpdateNotificator updateNotificator;

   public ThreadModel() {
      updateNotificator = new UpdateNotificator();
```

lec07/DemoBarComp-observer

Výpočetní model jako **Observable** objekt 2/4

- Musíme zajistit rozhraní pro přihlašování a odhlašování pozorovatelů
- Zároveň nechceme měnit typ výpočetního modelu ve třídě `MyBarPanel`
- Musíme proto rozšířit původní výpočetní model `Model`

```
public class Model {
 public void unregisterObserver(Observer observer) {...}
 public void registerObserver(Observer observer) {...}
 ...
```

- Ve třídě `ThreadModel` implementujeme přihlašování/odhlašování odběratelů

```
@Override
public void registerObserver(Observer observer) {
   updateNotificator.addObserver(observer);
}
@Override
public void unregisterObserver(Observer observer) {
   updateNotificator.deleteObserver(observer);
}
```

lec07/DemoBarComp-observer

Výpočetní model jako **Observable** objekt 3/4

- Odběratele informujeme po dílčím výpočtu v metodě `run` třídy `ThreadModel`

```
public void run() {
    ...
    while (!computePart() && !finished) {
        updateNotificator.update();
```

- Panel `MyBarPanel` je jediným odběratelem a implementuje rozhraní **Observer**, tj. metodu **update**

```
public class MyBarPanel extends JPanel implements
    Observer {
    @Override
    public void update(Observable o, Object arg) {
        updateProgress(); //arg can be further processed
    }
    private void updateProgress() {
        if (computation != null) {
            bar.setValue(computation.getProgress());
        }
    }
```

lec07/DemoBarComp-observer

Výpočetní model jako **Observable** objekt 4/4

- Napojení pozorovatele `MyBarPanel` na výpočetní model `Model` provedeme při nastavení výpočetního modelu

```
public class MyBarPanel extends JPanel implements
    Observer {
public void setComputation(Model computation) {
        if (this.computation != null) {
            this.computation.unregisterObserver(this);
        }
        this.computation = computation;
        this.computation.registerObserver(this);
    }
```

- Při změně modelu nesmíme zapomenout na odhlášení od původního modelu

  Nechceme dostávat aktualizace od původního modelu, pokud by dál existoval.

`lec07/DemoBarComp-observer`

Výpočetní vlákno ve Swing

- Alternativně můžeme využít třídu **SwingWorker**
- Ta definuje metodu **doInBackground()**, která zapouzdřuje výpočet na „pozadí" v samostatném vláknu
  - V těle metody můžeme publikovat zprávy voláním metody `publish()`
- Automaticky se také „napojuje" na události v „grafickém vlákně" a můžeme předefinovat metody
  - `process()` – definuje reakci na publikované zprávy
  - `done()` – definuje reakci po skočení metody `doInBackground()`

`http://docs.oracle.com/javase/tutorial/uiswing/concurrency/worker.html`

Příklad použití třídy SwingWorker 1/3

- Vlákno třídy `SwingWorker` využijeme pro aktualizaci GUI s frekvencí 25 Hz
- V metodě `doInBackground` tak bude periodicky kontrolovat, zdali výpočetní vlákno stále běží
- Potřebujeme vhodné rozhraní třídy `Model`, proto definujeme metodu `isRunning()`

```
public class Model {
    ...
    public boolean isRunning() { ... }
}
```

Není úplně vhodné, ale vychází z postupného rozšiřování původně nevláknového výpočtu. Lze řešit využitím přímo `ThreadModel`.

- Metodu `isRunning` implementujeme ve vláknovém výpočetním modelu `ThreadModel`

```
public class ThreadModel ...
    public synchronized boolean isRunning() {
        return thread.isAlive();
    }
```

`lec07/DemoBarComp-swingworker`

Příklad použití třídy SwingWorker 2/3

- Všechna ostatní rozšíření realizujeme pouze v rámci GUI třídy `MyBarPanel`
- Definujeme vnitřní třídy **MySwingWorker** rozšiřující **SwingWorker**

```
public class MyBarPanel extends JPanel {
    public class MySwingWorker extends SwingWorker<Integer
     , Integer> { ... }

    MySwingWorker worker;
```

- Tlačítko **Compute** připojíme k instanci `MySwingWorker`

```
private class ComputeListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        if (!worker.isDone()) { //only single worker
            status.setText("Start computation");
            worker.execute();
        }
    }
}
```

lec07/DemoBarComp-swingworker

Příklad použití třídy SwingWorker 3/3

- Ve třídě `MySwingWorker` definujeme napojení periodické aktualizace na *progress bar*

```
public class MySwingWorker extends SwingWorker {
    @Override
    protected Integer doInBackground() throws Exception {
        computation.compute();
        while (computation.isRunning()) {
            TimeUnit.MILLISECONDS.sleep(40); //25 Hz
            publish(new Integer(computation.getProgress()));
        }
        return 0;
    }
    protected void process(List<Integer> chunks) {
        updateProgress();
    }
    protected void done() {
        updateProgress();
    }
}
```

lec07/DemoBarComp-swingworker

- S výhodou využíváme přímého přístupu k `updateProgress`

Zvýšení interaktivity aplikace

- Po stisku tlačítka **Stop** aplikace čeká na doběhnutí výpočetního vlákna
- To nemusí být důvod k zablokování celého GUI
- Můžeme realizovat „vypnutí" tlačítek `Compute` a `Stop` po stisku `Stop`
- Jejich opětovnou aktivaci můžeme odložit až po ukočení běhu výpočetního vlákna

Diskutovaná témata

- Příklady vláken v GUI (Swing)
  - Návrhový vzor `Observer`
  - `SwingWorker`