Práce s více vlákny v knihovně Swing 6

V tomto semináři budeme demonstrovat, jak se vyhnout nepříjemnému „zasekávání" (lidově řečeno), které může nastat u dlouhotrvajících operací.

1 Motivace

Začneme motivačním příkladem. Předpokládejme, že máme jednoduchou aplikaci skládající se ze vstupního textového pole, do kterého můžeme zadat číslo n, a aplikace pro tuto hodnotu vypočte odpovídající Fibonacciho číslo.

Okno formuláře sestavíme běžným způsobem:

```
public class SwingFib extends JFrame {

    private JButton    btnStart;
    private JButton    btnStop;
    private JTextField txtInput;
    private JLabel     lblOutput;

    public SwingFib() {
        this.setTitle("Demo");
        this.setLayout(new FlowLayout());
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setSize(300, 70);

        txtInput = new JTextField("30", 5);
        lblOutput = new JLabel("Vysledek:");

        btnStart = new JButton("start");
        btnStop = new JButton("stop");
        btnStop.setEnabled(false);

        this.add(txtInput);
        this.add(lblOutput);
        this.add(btnStart);
        this.add(btnStop);
    }
```

```
}
```

Na tlačíko *Start* navážeme spuštění výpočtu:

```
btnStart.addActionListener(e -> {
    int arg = Integer.parseInt(txtInput.getText());
    int x = fib(arg);
    displayResult(x);
});
```

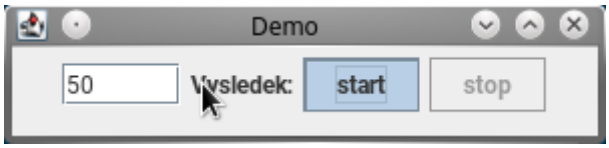
Metoda `int fib(int)` vypočte Fibonacciho číslo běžným (rekurzivním) způsobem.

```
public static int fib( int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n - 1) + fib(n - 2);
}
```

Metoda `void displayResult(int)` se postará o korektní zobrazení výsledku:

```
public void displayResult(int value) {
    lblOutput.setText(Integer.toString(value));
}
```

Pokud aplikaci spustíme a vyzkoušíme pro menší hodnoty (např. 10, 15, 20, . . .), bude se jevit, že funguje správně. Avšak pro větší hodnoty (např. 45, 50) aplikace přestává fungovat podle představ (minimálně běžných uživatelů). Jednotlivé komponenty nereagují (viz Obrázek 1), nedochází k překreslování okna. Výpočet nelze zastavit, protože na tlačítko `btnStop` není navázána žádná událost, která by byla s to výpočet přerušit. Jak by taková událost měla vypadat?



Obrázek 1: Okno v průběhu delšího výpočtu

Tento problém vzniká díky tomu, že všechny operace (včetně událostí), které jsou spojeny s grafickým uživatelským rozhraním, jsou zpracovány právě jedním vláknem. Toto vlákno se někdy označuje jako EventDispatchThread (zkráceně jen EDT). V idealizované podobě toto vlákno obsahuje smyčku typu:

```
while (msg = getMessage()) {
   switch (msg.type) {
     case WM_KEY:
        /* akce na stisk klavesy */
        break;
```

```
      case WM_MOUSE:
         /* akce spojena s aktivitou mysi */
         break;

      case WM_REPAINT:
         /* prekresleni okna */
         break;
      // ...
   }
}
```

V rámci této smyčky běžící program získává informace o vnějších událostech, které zahrnují, jak události vyvolané myší a klávesnící, tak i další systémové události, jako jsou např. požadavky na překreslení okna. Jednotlivé události jsou pak směrovány k odpovídajícím komponentám, které se postarají o reakci vyvoláním navázaných listenerů. Z toho plyne, pokud probíhá obsluha nějaké události (např. v rámci `ActionListeneru`), není možné provádět obsluhu dalších událostí. Pokud taková obsluha trvá delší dobu, aplikace se začne, lidově řečeno, „sekat".

Jako vhodné řešení se nabízí spustit déletrvající událost v samostatném vláknu. Má to však jeden háček. Knihovna Swing (podobně jako většina dalších knihoven pro uživatelské rozhraní) je navržena tak, že s grafickým rozhraním může pracovat jen EDT. Pokud zpracování události spustíme v odděleném vláknu, nelze tradičním způsobem zobrazit výsledek operace, protože není zajištěna (a ani není možné zajistit) synchronizaci mezi vláknem a EDT. V následujících kapitolách si ukážeme, jak tento problém překonat s využitím pomocné třídy `SwingWorker` a s vlastní implementací pomocí vláken.

2 Hotové řešení

Knihovna Swing pro řešení tohoto problému má nachystané řešení, kterým je třída `SwingWorker<T,V>`, která zajišťuje vytvoření samostatného vlákna a zároveň se stará o korektní promítnutí výsledku do uživatelského rozhraní.

Třída `SwingWorker<T,V>` má dvě klíčové metody:

(i) `protected T doInBackground() throws Exception`

(ii) `protected void done()`

Metoda `doInBackground()` definuje, jaký výpočet má probíhat na pozadí (v samostatném vlákně). Tato metoda nesmí nijak manipulovat s uživatelským rozhraním! K tomu slouží metoda `done()`, která je zavolána hned po skončení metody `doInBackground()`. V metodě `done()` již nejsme nijak omezování v tom, co můžeme dělat, třída `SwingWorker` se postará, aby kód byl vykonán v rámci EDT.

Ukažme si tedy použití v našem motivačním příkladu.

Vytvoříme potomka třídy `SwingWorker` jako vnořenou třídu našeho formuláře, to nám umožní pohodlné provázání s uživatelským rozhraním.

```
private class FibWorker extends SwingWorker<Integer, Void> {
    private final int arg;
    public FibWorker(int arg) {
        this.arg = arg;
    }
    @Override
    protected Integer doInBackground() throws Exception {
        return fib(arg);
    }
    @Override
    protected void done() {
        try {
            displayResult(this.get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

Třída `SwingWorker<T,V>` má dva typové parametry. Parametr `T` udává typ hodnoty vrácené metodou `doInBackground()`, parametr `V` se používá při informování o průběhu výpočtu. Tuto funkcionalitu nevyužíváme, proto použijeme typ `Void`.

V metodě `doInBackground()` je vyvolán výpočet, který má proběhnout na pozadí. V tomto případě voláme výpočet Fibonacciho čísla pro hodnotu, která byla předána konstruktoru. Když doběhne výpočet, získáme v metodě `done()` jeho výsledek zavoláním metody `SwingWorker.get()` a zobrazíme jej metodou `displayResult(int)` podobně jako v původní verzi aplikace. Metoda `SwingWorker.get()` může skončit výjimkou, proto je potřeba ji nějakým vhodným způsobem ošetřit. Vhodnější by bylo použít dialogové okno, ale v tomto případě si to zjednodušíme a vypíšeme výjimku na terminál.

Dále upravíme chování tlačítka *Start* tak, aby využívalo nově vytvořené třídy `FibWorker`. Nejdříve vytvoříme atribut `fibWorker`:

```
private FibWorker fibWorker;
```

V akci svazané s tlačítkem načteme hodnotu ze vstupu, vytvoříme instanci třídy `FibWorker` s odpovídajícím argumentem a zahájíme výpočet zavoláním metody `FibWorker.execute()`.

```
btnStart.addActionListener(e -> {
    int arg = Integer.parseInt(txtInput.getText());
    fibWorker = new FibWorker(arg);
    fibWorker.execute();
});
```

Když si program nyní vyzkoušíme, zjistíme, že hlavní problém „zasekávání" se nám podařilo úspěšně

vyřešit. Program reaguje i v průběhu déletrvajícího výpočtu, avšak toto řešení není zcela uspokojivé a to ze tří důvodů:

(i) uživatel není informován o tom, že probíhá výpočet;

(ii) je možné opětovně zmáčknout tlačítko *Start* (pokud nějaký výpočet trvá delší dobu, uživatelé budou mít tendenci toto tlačítko mačkat opakovaně v marné naději, že výpočet doběhne);

(iii) není možné déletrvající výpočet přerušit.

Problémy (i) a (ii) lze vyřešit tím, že před spuštěním výpočtu upravíme uživatelské rozhraní, aby bylo vidět, že výpočet probíhá, znemožníme jeho násobné spuštění, případně povolíme tlačíko pro přerušení výpočtu:

```
lblOutput.setText("Wait!");
btnStart.setEnabled(false);
btnStop.setEnabled(true);
```

Současně je nutné po skončení výpočtu vrátit uživatelského rozhraní do původního stavu, např. v našem případě pomocí úpravy metody `displayResult(int)`.

```
public void displayResult(int value) {
    lblOutput.setText(Integer.toString(value));
    btnStart.setEnabled(true);
    btnStop.setEnabled(false);
}
```

Přerušení výpočtu lze realizovat pomocí metody `SwingWorker.cancel(boolean)`, kterou můžeme navázat na tlačítko *Stop*.

```
btnStop.addActionListener(e -> fibWorker.cancel(true));
```

Pozor! Tato metoda sama od sebe nepřeruší výpočet v metodě `doInBackground`[1], ale spíše signalizuje, že byl vyslán požadavek na přerušení výpočtu a metoda `doInBackground()` by na to měla adekvátně zareagovat. Proto upravíme kód třídy `FibWorker`, aby se s takovým požadavkem dokázala vypořádat. V našem případě vložíme do výpočtu Fibonacciho čísla test, zda nepřišel požadavek na přerušení výpočtu, a pokud ano, výpočet ukončíme.

```
public static int fib(SwingWorker<Integer, Void> w, int n) {
    if (w.isCancelled()) return -1;
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(w, n - 1) + fib(w, n - 2);
}
```

[1]Metoda by mohla mít otevřené soubory, spojení do databáze, držet zámky, apod., a kdyby došlo k náhlému přerušení výpočtu, mohl by se program dostat do konzistentního stavu nebo deadlocku.

Metodu `doInBackground()` upravíme, aby předávala do výpočtu Fibonacciho čísla i odkaz na svou instanci.

```
@Override
protected Integer doInBackground() throws Exception {
    return fib(this, arg);
}
```

Nakonec ještě uprávíme metodu `done()`, aby zobrazila informaci o přerušení výpočtu.

```
protected void done() {
    try {
        if (this.isCancelled()) displayCancel();
        else displayResult(this.get());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Metoda `displayCancel()` je velmi podobná metodě `displayResult(int)`, jejím smyslem je informovat o nedokončeném výpočtu a vrátit uživatelské rozhraní do původního stavu.

```
public void displayCancel() {
    lblOutput.setText("Cancelled");
    btnStart.setEnabled(true);
    btnStop.setEnabled(false);
}
```

3 Explicitní práce s vlákny

Řešení představené v předchozí kapitole pokrývá nejtypičtější scénář, kdy potřebujeme pracovat s grafickým rozhraním a současně s více vlákny, tj. úloha se spustí na pozadí a nakonec se zobrazí výsledek. Tento scénář nemusí být vhodný pro všechny situace a dále, abychom si ukázali, jak `SwingWorker` pracuje pod pokličkou, ukážeme si, jak celý problém uchopit s explicitním použitím vláken.

Vytvoříme třídu reprezentující samostatné vlákno počítající Fibonacciho číslo.

```
private class FibThread extends Thread {
    private int n;
    public FibThread(int n) {
        super();
        this.n = n;
    }
```

```
    @Override
    public void run() {
        int result = fib(n);
        // FIXME: nelze zavolat
        // displayResul(result);
    }
}
```

Zda narážíme na problém, že nelze měnit grafické uživatelské rozhraní z jiného vlánka (tj. zobrazit výsledek). To se však dá překonat s pomocí metody `SwingUtilities.invokeLater(Runnable)`, která se postará, aby kód, který je jí předán v argumentu, byl zavolán v rámci vlákna EDT. Metoda `run()` by po této změně měla vypadat:

```
public void run() {
    int result = fib(n);
    SwingUtilities.invokeLater(() -> displayResult(result));
}
```

Nyní už máme kód, který lze v základní variantě zakomponovat do našeho motivačního příkladu.

Vytvoříme atribut pro vlákno:

```
private FibThread thr;
```

Na tlačíko *Start* navážeme vytvoření a spuštení vlákna:

```
btnStart.addActionListener(e -> {
    int arg = Integer.parseInt(txtInput.getText());
    adjustGUI(); // upravi uzivatelske rozhrani
    thr = new FibThread(arg);
    thr.start();
});
```

Řešení je to však pouze částečné, protože zatím neumožňuje přerušení výpočtu.

Rozšíříme proto třídu `FibThread` o další attribut

```
private boolean cancelled;
```

signalizující požadavek na přerušení výpočtu. S tímto atributem se bude manipulovat pomocí dvou metod:

```
public synchronized boolean isCancelled() {
    return cancelled;
}
public synchronized void cancel() {
    this.cancelled = true;
}
```

Obě metody by měly být deklarovány jako `synchronized`, protože se s nimi bude přistupovat k objektu ze dvou různých vláken, tj. z EDT a ze samotného vlákna provádějícího výpočet.

Metodu pro výpočet Fibonacciho čísla upravíme, aby byla schopna zaregovat na přerušení výpočtu, pokud zjistí, že vláknu byl nastavený atribut `cancelled` na `true`.

```
private int fib(FibThread t, int n) {
    if (t.isCancelled()) return -1;
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(t, n - 1) + fib(t, n - 2);
}
```

Současně adekvátně upravíme i metodu `FibThread.run()`.

```
public void run() {
    final int result = fib(this, n);
    SwingUtilities.invokeLater(() -> {
        if (isCancelled()) displayCancel();
        else displayResult(result);
    });
}
```

Závěrem stačí upravit událost navazánou na tlačítko *Stop*.

```
btnStop.addActionListener(e -> thr.cancel());
```

Všimněme si, že kód používající `SwingWorker` a vlákna se příliš neliší. V druhém případě si jen programátor musí hlídat použití `SwingUtilities.invokeLater(Runnable)` k provádění změn v uživatelském rozhraní z jiného vlákna než EDT a zároveň musí mít (pomocí `synchronized`) ošetřený přístup k datům, se kterými pracuje více vláken současně.