

Vícevláknové aplikace – modely a příklady

Jiří Vokřínek

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 8

B0B36PJV – Programování v JAVA

Část 1 – Využití vláken v GUI

Vlákna v GUI (Swing)

Rozšíření výpočetního modulu v aplikaci DemoBarComp o vlákno

Využití třídy SwingWorker

Část 2 – Modely vícevláknových aplikací

Modely více-vláknových aplikací

Prostředky ladění

Část 3 – Příklad – GUI aplikace Simulátor/Plátno

GUI s plátnem

- Struktura aplikace

- Struktura simulátoru

- Struktura grafického rozhraní

- Praktické ukázky

Část I

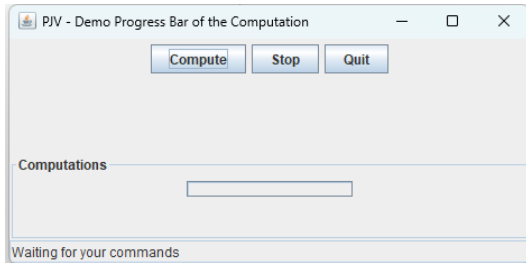
Část 1 – Využití vláken v GUI

Příklad aplikace – BarComp

Naším cílem je vytvořit jednoduchou aplikaci se sadou tlačítek pro ovládání výpočtu, s vizualizací postupu výpočtu a stavu aplikace.

■ Aplikace má 3 základní komponenty

1. Hlavní ovládací tlačítka
2. „Progress bar”
3. Stavový řádek



Struktura aplikace – BarComp

- Aplikace se skládá z výpočetního modelu `Model`, grafických komponent `MyBarPanel` a spouštěcí třídy `DemoBarComp`

```
public class DemoBarComp {
```

```
    void start() { ... }
```

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

```
        DemoBarComp demo = new DemoBarComp();
```

```
        demo.start();
```

```
    }
```

```
}
```

`lec08/DemoBarCompNew`

Struktura aplikace – BarComp

- Aplikace se skládá z výpočetního modelu `Model`, grafických komponent `MyBarPanel` a spouštěcí třídy `DemoBarComp`

```
public class DemoBarComp {  
  
    void start() { ... }  
  
    public static void main(String[] args) {  
        DemoBarComp demo = new DemoBarComp();  
        demo.start();  
    }  
}
```

`lec08/DemoBarCompNew`

Struktura aplikace – DemoBarComp – start

```
void start() {  
    JFrame frame = new JFrame("PJV - Demo Progress Bar  
    of the Computation");  
    frame.setDefaultCloseOperation(JFrame.  
    EXIT_ON_CLOSE);  
    frame.setMinimumSize(new Dimension(480, 240));  
  
    MyBarPanel myBarPanel = new MyBarPanel();  
  
    frame.getContentPane().add(myBarPanel);  
    frame.pack();  
    frame.setVisible(true);  
  
    myBarPanel.setComputation(new Model());  
}
```

lec08/DemoBarCompNew/DemoBarComp.java

MyBarPanel – základní struktura

```
public class MyBarPanel extends JPanel {  
    JTextField status;  
    JProgressBar bar;  
  
    Model computation;  
  
    public MyBarPanel() {  
        computation = null;  
        createComponents();  
    }  
    public void setComputation(Model computation) {  
        this.computation = computation;  
    }  
    private void createComponents() { ... }  
}
```

lec08/DemoBarCompNew/MyBarPanel.java

MyBarPanel – základní struktura

- `createComponents` – vytvoří odpovídající komponenty GUI
- `createControlButtons` – vytvoří tři ovládací tlačítka a přiřadí k nim instance vnitřních tříd příslušných posluchačů (`addActionListener`)
 - `ComputeListener` – spustí metodu Modelu `compute`
 - `StopListener` – zatím nic nedělá
 - `QuitListener` – ukončí aplikaci voláním `System.exit(0)`
- `createProgress` – vytvoří `progressBar` pro vizualizaci postupu výpočtu
- `createStatusBar` – vytvoří textové pole pro prezentaci stavu aplikace

`lec08/DemoBarCompNew/MyBarPanel.java`

Model – základní struktura

- slouží k simulaci výpočtu
- `computeSinglePiece` – metoda simulující 100ms výpočtu pomocí uspání vlákna
- `compute(MyBarPanel panel)` – metoda 100x volá `computeSinglePiece` ve dvojitém cyklu
 - v každé desáté iteraci vypisuje stav výpočtu na obrazovku a
 - zavolá pomocí *call-backu aktualizaci GUI* `panel.updateProgress`

`lec08/DemoBarCompNew/Model.java`

Příklad aplikace – BarComp – nevýhody

Aplikace obsahuje všechnu potřebnou funkcionalitu, ale zatím nefunguje správně.

- Výpočet probíhá ve vlákně obsluhy tlačítka
- Hodnota postupu výpočtu se zapisuje do správné komponenty, ale průběžně se nezobrazuje

Proč?

- Aplikace není dostatečně interaktivní

Aplikaci opravíme přesunutím výpočtu do samostatného vlákna

Příklad aplikace – BarComp – nevýhody

Aplikace obsahuje všechnu potřebnou funkcionalitu, ale zatím nefunguje správně.

- Výpočet probíhá ve vlákně obsluhy tlačítka
- Hodnota postupu výpočtu se zapisuje do správné komponenty, ale průběžně se nezobrazuje

Proč?

- Aplikace není dostatečně interaktivní

Aplikaci opravíme přesunutím výpočtu do samostatného vlákna

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

- Třidu `Model` upravíme pro spouštění v samostatném vlákně
 - Metoda `compute` se bude spouštět v samostatném vlákně
 - Musíme zabránit opakovanému spuštění

Příznak `running`

- Metodu uděláme synchronizovanou
- Po stisku tlačítka stop ukončíme vlákno

Implementujeme třídu `StopListener`

- Ve třídě `Model` implementuje metodu `stop`

Nastaví příznak ukončení výpočetní smyčky `stop`

`lec08/DemoBarCompNewSimpleThread`

Po spuštění výpočtu je GUI aktivní, *progress bar* se aktualizuje (*push*) notifikací z nového vlákna

Výpočetní vlákno ve Swing

- Alternativně můžeme využít třídu `SwingWorker`
- Ta definuje metodu `doInBackground()`, která zapouzdřuje výpočet na „pozadí“ v samostatném vlákně
 - 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` 1/2

- Vlákno třídy `SwingWorker` využijeme pro aktualizaci GUI s frekvencí 25 Hz
- V metodě `doInBackground` tak bude periodicky aktualizovat stav výpočtu `publish(computation.getProgress())` nezávisle na běhu výpočetního vlákna
- Všechna ostatní rozšíření realizujeme pouze v rámci GUI třídy `MyBarPanel`
- Definujeme vnitřní třídu `MySwingWorker` rozšiřující `SwingWorker`
- Nebudeme již potřebovat předávat metodě `compute` referenci na `MyBarPanel`

`lec08/DemoBarCompNewSwingWorker`

Příklad použití třídy `SwingWorker` 2/2

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

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

lec08/DemoBarCompNewSwingWorker

Zvýšení interaktivity aplikace

- Aplikace `DemoBarComp` je jednoduchou ukázkou interaktivní aplikace
- Lze vylepšit vyšší interakcí s výpočetním vláknem a definování komplexnějších stavů výpočtu
- Můžeme realizovat „vypnutí“ tlačítek `Compute` a `Stop` po stisku `Stop`
- Jejich opětovnou aktivaci můžeme odložit až po ukočení běhu výpočetního vlákna

Část II

Část 2 – Modely vícevláknových aplikací

Kdy použít vlákna?

Vlákna je výhodné použít, pokud aplikace splňuje některé následující kritérium:

- Je složena z nezávislých úloh.
- Může být blokována po dlouho dobu.
- Obsahuje výpočetně náročnou část.
- Musí reagovat na asynchronní události.
- Obsahuje úlohy s nižší nebo vyšší prioritou než zbytek aplikace.

Typické aplikace

- **Servery** - obsluhují více klientů najednou. Obsluha typicky znamená přístup k několika sdíleným zdrojům a hodně vstupně výstupních operací (I/O).
- **Výpočetní aplikace** - na víceprocesorovém systému lze výpočet urychlit rozdělením úlohy na více procesorů.
- **Aplikace reálného času** - lze využít specifických rozvrhovačů. Vícevláknová aplikace je výkonnější než složité asynchronní programování, neboť vlákno čeká na příslušnou událost namísto explicitního přerušování vykonávání kódu a přepínání kontextu.

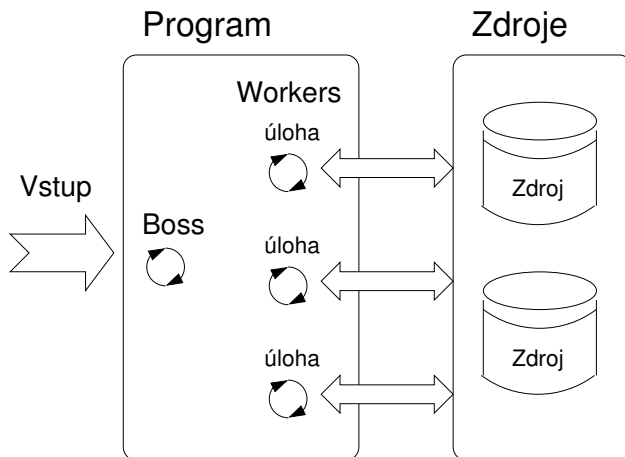
Modely vícevláknových aplikací

Modely řeší způsob vytváření a rozdělování práce mezi vlákny.

- **Boss/Worker** - hlavní vlákno řídí rozdělení úlohy jiným vláknům.
- **Peer** - vlákna běží paralelně bez specifického vedoucího.
- **Pipeline** - zpracování dat sekvencí operací.

Předpokládá dlouhý vstupních proud dat.

Boss/Worker model



Boss/Worker rozdělení činnosti

- Hlavní vlákno je zodpovědné za vyřizování požadavků.
- Pracuje v cyklu:
 1. příchod požadavku,
 2. vytvoření vlákna pro řešení příslušného úkolu,
 3. návrat na čekání požadavku.
- Výstup řešení úkolu je řízen:
 - Příslušným vláknem řešícím úkol.
 - Hlavním vláknem, předání využívá synchronizační mechanismy.

Boss/Worker příklad

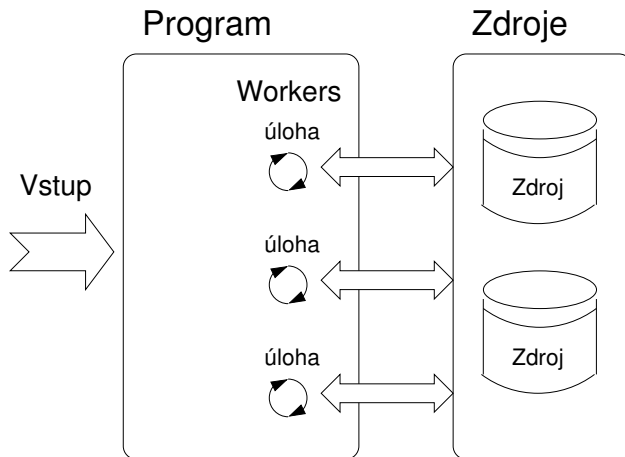
Příklad Boss/Worker model

```
1 //Boss
2 main() {
3     while(1) {
4         switch(getRequest()) {
5             case taskX :
6                 create_thread(taskX);
7             case taskY :
8                 create_thread(taskY);
9             :
10        }
11    }
```

```
1 //Worker
2 taskX() {
3     řešení úlohy,
4     synchronizace v
5     případě sdílených
6     zdrojů;
7     done;
8 }
9 taskY() {
10    řešení úlohy,
11    synchronizace v
12    případě sdílených
13    zdrojů;
14    done;
15 }
```

C style

Peer model



Peer model - vlastnosti

- Neobsahuje hlavní vlákno.
- První vlákno po vytvoření ostatních vláken:
 - se stává jedním z ostatních vláken (rovnocenným),
 - pozastavuje svou činnost do doby než ostatní vlákna končí.
- Každé vlákno je zodpovědné za svůj vstup a výstup.

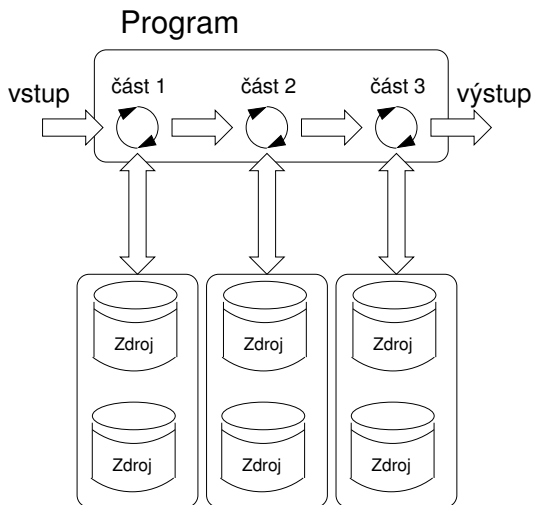
Peer model - příklad

Příklad Peer model

```
1 //Boss
2 main() {
3     create_thread(task1);
4     create_thread(task2);
5     :
6     :
7     start all threads;
8     wait for all threads;
9 }
```

```
1 //Worker
2 task1() {
3     čekáná na spuštění;
4     řešení úlohy,
5     synchronizace v
6     případě sdílených
7     zdrojů;
8     done;
9 }
10 task2() {
11     čekáná na spuštění;
12     řešení úlohy,
13     synchronizace v
14     případě sdílených
15     zdrojů;
16     done;
17 }
```

Zpracování proudu dat - Pipeline



Pipeline

- Dlouhý vstupní proud dat.
- Sekvence operací (částí zpracování), každá vstupní jednotka musí projít všemi částmi zpracování.
- V každé části jsou v daném čase, zpracovávány různé jednotky vstupu (nezávislost jednotek).

Pipeline model - příklad

Příklad Pipeline model

```
1  main() {
2      create_thread(stage1);
3      create_thread(stage2);
4      :
5      :
6      wait for all pipeline;
7  }
8  stage1() {
9      while(input) {
10         get next program
            input;
11         process input;
12         pass result to next
            stage;
13     }
14 }
```

```
1  stage2() {
2      while(input) {
3         get next input from
            thread;
4         process input;
5         pass result to next
            stage;
6     }
7  }
8  stageN() {
9      while(input) {
10         get next input from
            thread;
11         process input;
12         pass result to output;
13     }
14 }
```

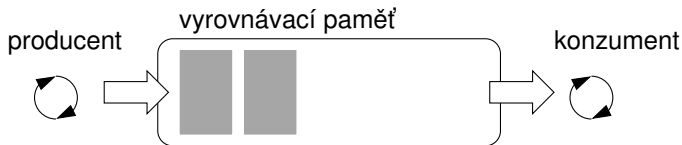
C style

Producent a konzument

Předávání dat mezi vlákny je realizováno vyrovnávací pamětí bufferem.

- Producent - vlákno, které předává data jinému vláknu.
- Konzument - vlákno, které přijímá data od jiného vlákna.

Přístup do vyrovnávací paměti musí být synchronizovaný (exkluzivní přístup).



Funkce a paralelismus

Při paralelním běhu programu mohou být funkce volány vícenásobně. Funkce jsou :

- **reentrantní** - V jediném okamžiku může být stejná funkce vykonávána současně

Např. vnořená obsluha přerušení

- **thread-safe** - Funkci je možné současně volat z více vláken

Dosažení těchto vlastností:

- Reentrantní funkce nezapisují do statických dat, nepracují s globálními daty.
- Thread-safe funkce využívají synchronizačních primitiv při přístupu ke globálním datům.

Vícevláknové aplikace a ladění

Hlavní problémy vícevláknových aplikací souvisí se synchronizací:

- **Uvážnutí – deadlock.**
- **Souběh (race conditions)** - přístup více vláken ke sdíleným proměnným a alespoň jedno vlákno nevyužívá synchronizačních mechanismů. Vlákno čte hodnotu zatímco jiné vlákno zapisuje. Zápis a čtení nejsou atomické a data mohou být neplatná.

Prostředky ladění

- Nejlepším prostředkem ladění vícevláknových aplikací je **nepotřebovat ladit.**
- Toho lze dosáhnout kázní a obezřetným přístupem ke sdíleným proměnným.
- Nicméně je vhodné využívat ladící prostředí s minimální množinou vlastností.

Prostředky ladění

- Nejlepším prostředkem ladění vícevláknových aplikací je **nepotřebovat ladit.**
- Toho lze dosáhnout kázní a obezřetným přístupem ke sdíleným proměnným.
- Nicméně je vhodné využívat ladící prostředí s minimální množinou vlastností.

Prostředky ladění

- Nejlepším prostředkem ladění vícevláknových aplikací je **nepotřebovat ladit.**
- Toho lze dosáhnout kázní a obezřetným přístupem ke sdíleným proměnným.
- Nicméně je vhodné využívat ladící prostředí s minimální množinou vlastností.

Podpora ladění

Debugger:

- Výpis běžících vláken.
- Výpis stavu synchronizačních primitiv.
- Přístup k proměnným vláken.
- Pozastavení běhu konkrétního vlákna.
- Záznam průběhu běhu celého programu (kompletní obsah paměti a vstupů/výstup) a procházení záznamu

Logování:

- Problém uváznutí souvisí s pořadím událostí, logováním přístupu k zámekům lze odhalit případné špatné pořadí synchronizačních operací.

Poznámky - „problémy souběhu“

Problémy souběhu jsou typicky způsobeny nedostatkem synchronizace.

- Vlákna jsou **asynchronní**.

Nespoléhat na to, že na jednoprocessorovém systému je vykonávání kódu synchronní.

- Při psaní vícevláknové aplikace předpokládejte, že vlákno může být kdykoliv přerušeno nebo spuštěno.

Části kódu, u kterých je nutné zajistit pořadí vykonávání jednotlivými vlákny vyžadují synchronizaci.

- Nikdy nespolehejte na to, že vlákno po vytvoření počká, může být spuštěno velmi brzy.

- Pokud nespecifikujete pořadí vykonávání vláken, žádné takové neexistuje.

„Vlákna běží v tom nejhorším možném pořadí. Bill Gallmeister“

Poznámky - „problém uváznutí“

Problémy uváznutí souvisí s mechanismy synchronizace.

- Uváznutí (deadlock) se na rozdíl o souběhu mnohem lépe ladí.
- Častým problémem je tzv. *mutex deadlock* způsobený pořadím získávání mutexů (zámků/monitorů).
- Mutex deadlock nemůže nastat, pokud má každé vlákno uzamčený pouze jeden mutex (*chce uzamknout*).
- Není dobré volat funkce s uzamčeným mutexem, obzvláště zamyká-li volaná funkce jiný mutex.
- Je dobré zamykat mutex na co možná nejkratší dobu.

V Javě odpovídá zámek kříčkové sekci monitoru `synchronized(mtx){}`

[http://www.javaworld.com/article/2076774/java-concurrency/
programming-java-threads-in-the-real-world--part-1.html](http://www.javaworld.com/article/2076774/java-concurrency/programming-java-threads-in-the-real-world--part-1.html)

Část III

Část 3 – Příklad – GUI aplikace Simulator/Plátno

Zadání

- Naším cílem je vytvořit simulátor „herního“ světa
 - Ve světě mohou být různé objekty, které se mohou nezávisle pohybovat
 - Simulátor je „nezávislý“ na vizualizaci
 - Simulátor běží v diskrétních krocích
- Vizualizaci herního světa se pokusíme oddělit od vlastního simulátoru
 - Každému objektu simulátoru přiřadíme grafický tvar, který se bude umět zobrazit na plátno
- Simulátor chceme ovládat tlačítky „Start/Stop“
- Svět simulátoru vizualizujeme na plátně
- Vizualizace plátna bude probíhat „nezávisle“ na běhu simulátoru

Interaktivní hra vs Simulace

Základní struktura aplikace

■ Simulátor – obsahuje svět a objekty

- V zásadě se chová jako kolekce simulačních objektů

Iterable

- Simulace běží v samostatném vlákně s periodou `PERIOD`
- Simulace probíhá v diskrétních časových okamžicích voláním metody `doStep` simulačních objektů
- Má metodu pro zastavení vlákna `shutdown`

■ Grafické rozhraní a vizualizace – obsahuje

- Základní kontrolní tlačítka pro ovládání simulace (start/stop)
- Plátno pro vykreslení dílčích objektů
- Standardní vykreslovací Swing vlákno
- Samostatné vlákno pro překreslování stavu simulátoru

`SwingWorker` s přeposíláním zpráv hlavnímu Swing vláknu

- Grafickou reprezentaci vykreslovaných objektů

Vlastní vykreslení grafickými primitivy.

Simulator – SimObject

- **Simulator** – kolekce simulačních objektů
- **SimObject** – jednotné rozhraní simulačního objektu
 - Aktuální pozice objektu – `public Coords getPosition();`
 - Provedení jednoho simulačního kroku – objekt má definované chování `public void doStep();`
- Simulace probíhá ve smyčce:

```
while(!quit) {  
    for(SimObject obj : objects) {  
        obj.doStep();  
    }  
    Thread.sleep(PERIOD);  
}
```

Struktura grafického rozhraní

- Hlavní okno aplikace obsahuje kontrolní tlačítka
 - Tlačítko `start` spouští simulaci
 - Tlačítko `stop` zastavuje běžící simulaci
- Vizualizace simulace probíhá ve vlastním plátně `MyCanvas`
- Simulační objekty mají svůj grafický tvar `Shape`
- Překreslení plátna probíhá periodicky

`SwingWorker()`

```
while(sim.isRunning()) {  
    if (sim.isChanged()) {  
        MyCanvas canvas = getSimCanvas();  
        canvas.redraw();  
        Thread.sleep(CANVAS_REFRESH_PERIOD);  
    }  
}
```

Základní koncept překreslení – neodpovídá kódu

Struktura plátna `MyCanvas` a vizualizace

- `MyCanvas` – obsahuje kolekci vykreslitelných objektů – instance `Shape`
- Každý objekt se umí vykreslit do grafického kontextu

```
public void redraw() {  
    Graphics2D g2d = getGraphics();  
    for (Shape obj : shapes) {  
        obj.draw(g2d);  
    }  
}
```

- Vlastní tvar objektu je definován v konkrétní třídě implementující `Shape`

```
public class SimObjectMonster extends SimObject  
    implements Shape {  
    ...  
}
```


Příklad definice tvaru

■ SimObjectMonster

```
public class SimObjectMonster extends SimObject implements
    Shape {
    public SimObjectMonster(Dimension dimension, Coords pose) {
        super(dimension,pose,Direction.RIGHT,2);
    }
    @Override
    public void draw(Graphics2D g2d) {
        Coords pt = getPosition();
        g2d.setColor(Color.RED);
        g2d.fillOval(pt.getX(), pt.getY(), 15, 15);
    } }
}
```

■ SimObjectNPC

```
public class SimObjectNPC extends SimObject implements Shape {
    public SimObjectNPC(Dimension dimension, Coords pose) {
        super(dimension,pose,Direction.UP,1);
    }
    @Override
    public void draw(Graphics2D g2d) {
        Coords pt = getPosition();
        g2d.setColor(Color.GREEN);
        g2d.fillRect(pt.getX(), pt.getY(), 15, 15);
    } }
}
```

Vytvoření simulačních objektů a jejich tvarů

```
void start() {  
    ...  
    Simulator sim = new Simulator();  
    MyPanel myPanel = new MyPanel(sim);  
    createSimObjects(sim, myPanel);  
    ...  
}  
  
void createSimObjects(Simulator sim, MyPanel gui) {  
    Dimension world = sim.getWorldDimensions();  
  
    SimObjectMonster monster = new SimObjectMonster(world  
        , new Coords(10, 10));  
    sim.addSimObject(monster);  
    gui.addShapeObject(monster);  
  
    SimObjectNPC npc = new SimObjectNPC(world, new Coords  
        (400, 200));  
    sim.addSimObject(npc);  
    gui.addShapeObject(npc);  
}
```

Příklad – CanvasDemo

- Překreslování prostřednictvím `SwingWorker` vs přímé překreslování ve vlastním vlákně
- `DoubleBuffer` – přepínání vykresleného obrazu
- Časování a zajištění periody
- Plynulé překreslování bez pohybu myši

```
Toolkit.getDefaultToolkit().sync();
```

`lec08/CanvasDemo`

Simulace vs grafická hra

- V simulaci se zpravidla snažíme důsledně oddělit simulované objekty od vizualizace

1. Na úrovni simulačních objektů a jejich vizuální reprezentace
2. Na úrovni simulačního času a rychlosti překreslování

Přesnost simulace má přednost před rychlou a včasnou vizualizací (v reálném čase)

- Hry jsou zpravidla silně svázány s grafickou vizualizací

- Krok herního světa zpravidla znamená překreslení
- Kreslící vlákno tak udává také simulační krok
- Klíčovým aspektem je zachování plynulosti vizualizace a interakce

V případě pomalejšího překreslování je rychlost herního světa adekvátně zpomalena.

- Interaktivní hry zpravidla využívají individuálního kreslení objektů do plátna (případně 3D kontextu)

Používají vlastní sadu komponent (widgets), zpravidla vizuálně efektní, princip je však stejný jako například ve Swing.

Chceme-li maximalizovat využití zdrojů a zajistit vysokou interaktivitu zpravidla musíme mít plně pod kontrolou běh aplikace.

Shrnutí přednášky

Diskutovaná témata

- Využití vláken v GUI
- Modely vícevláknových aplikací
- Paralelní programování a ladění
 - Problém uváznutí a problém souběhu
- GUI plátno – simulátor a kreslení do canvasu

- Příště: Sokety a síťování