

# KONKURENTNÉ PROGRAMOVANIE

Cvičenie 9 : JavaFx

# Používateľ chce svižné GUI

## □ Responsiveness

### ▣ Okamžitá ~ 100ms

- Klik na tlačidlo spôsobí okamžité stlačenie

### ▣ Bezprostredná ~ 0,5s – 1s

- Doba medzi odoslaním a prijatím príkazu

### ▣ Priebežná ~ 2s – 5s

- Perióda informovanosti o prograse

### ▣ Pútavá ~ do 10s

- Maximálna doba pozornosti používateľ'a

# Typicky pomalé operácie

- Sieťové operácie
  - ▣ Sokety, ťahanie z internetu
- Kopírovanie súborov
- Práca s databázou
- Spracovanie videa
- Refresh zdrojov

Všetko nad  
500 ms

# Princíp fungovania aplikácii s GUI

- Vlákno JavaFX Application thread
  - ▣ Spracovanie udalostí v GUI
    - Klik na button, posun slidera, pozícia myšky, výber položky, editácia textu, ...
    - Všetky listenery komponentov
  - ▣ Zmena vzhľadu GUI
    - Pridanie a odobratie komponentov
  - ▣ Refresh GUI
    - Vizualizácia zmeny vlastností komponentov (posun progress baru, zvýraznenie vybraného komponentu,...)
- Na všetko ostatné iné vlákna

# Princíp fungovania aplikácii s GUI

- JavaFX Application thread má rad úloh
  - ▣ Úlohy na zmenu GUI, spracovanie udalostí komponentov
  - ▣ Vykonáva ich postupne
  - ▣ Nové úlohy na koniec
- V JAT beží nekonečný cyklus
  - ▣ Vyberie úlohu z radu
  - ▣ Vykoná ju
  - ▣ Prekreslí GUI

# Zásada práce v okienkovej apke

- V JAT nevykonávajte dlhotrvajúce operácie
  - ▣ Dlhotrvajúce úlohy blokujú rad úloh pre GUI
  - ▣ Používateľ má pocit, že aplikácia vytuhla
    - začne zbesilo klikať, lenže tým si nepomôže, lebo iba generuje udalosti (úlohy) radené na koniec radu!
- Stav GUI komponentov a ich modelov nemeňte inde než v JAT
  - ▣ Komponenty, XXXProperty a Observable väčšinou nie sú thread-safe
  - ▣ Viac vlákien by mohlo volať metódy toho istého komponentu a spôsobiť nekonzistentnosť jeho stavu, deadlocky, zvláštne chovanie,..

# Dlhé úlohy iniciované v JAT

- Ak chceme v JAT iniciovať dlhý výpočet, musíme ho spustiť v samostatnom vlákne (Thread),
- Runnable úlohy si však žijú vlastným životom
  - ▣ Nevedia nám vrátiť výsledok
  - ▣ Nevieme ich zrušiť
  - ▣ Nevieme odchytiť výnimky

# Task: „Observable“ úloha

- Trieda **Task**
  - Úloha pre vykonávanie mimo JAT
  - Vyrábame potomka, ktorý pokrýva metódu **call()**, ktorá pobeží v novom, tzv. **worker** vlákne
  - Task môže vygenerovať niekoľko udalostí, na ktoré môžeme zaregistrovať vlastné listenery
  - Listenery sú automaticky spúšťané v JAT vlákne



# Obslužné úlohy

- metódy na registráciu obslužných úloh:
  - `setOnScheduled(event)`
    - Ked' sa úloha naplánuje
  - `setOnRunning(event)`
    - Ked' sa úloha začne vykonávať
  - `setOnCancelled(event)`
    - Ked' úlohu zrušíme volaním `cancel()` na `task-u`
  - **`setOnFailed(event)`**
    - Ked' úloha skončí s výnimkou
  - **`setOnSucceeded(event)`**
    - Ked' úloha vráti hodnotu

# Minimalistický príklad

```
Task<Void> myTask = new Task<Void>() {  
    protected Void call() throws Exception { // vykoná sa v novom vlákne  
        zaplňCelýDiskSomarinami();  
        return null;  
    }  
};  
  
Thread th = new Thread(myTask);  
th.start();
```

- `call()` musí niečo vrátiť
  - ak nemáme čo, uvedieme ako návratový typ `java.lang.Void`
  - vrátíme `null`

# Príklad s návratovou hodnotou

## □ Zmena modelu v GUI po skončení úlohy:

```
Task<String> meaningTask = new Task<String>() {  
    protected String call() throws Exception {  
        return findTheMeaningOfLife();  
    }  
};
```

spustené vo  
worker vlákne

```
meaningTask.setOnSucceeded(new EventHandler<WorkerStateEvent>() {  
    public void handle(WorkerStateEvent event) {  
        String theMeaning = meaningTask.getValue();  
        meaningFxModel.setMeaning(theMeaning);  
    }  
});
```

spustené v  
JAT vlákne

```
Thread th = new Thread(meaningTask);  
th.start();
```

# Odchytenie výnimky

- Ak worker vlákno skončí s výnimkou, vieme na túto udalosť reagovať listenerom v JAT

```
Task<Void> myTask = new Task<Void>() {  
    protected Void call() throws Exception { // vykoná sa v novom vlákne  
        throw new MyWorkerException();  
    }  
};
```

```
myTask.setOnFailed(new EventHandler<WorkerStateEvent>() {  
    public void handle(WorkerStateEvent event) {  
        Throwable throwable = myTask.getException();  
        throwable.printStackTrace();  
    }  
});
```

# Service – Spúšťáč tasku

- Vieme na ňom odchytať tie isté udalosti, ako na Tasku

```
Service<String> meaningService = new Service<String>() {  
    protected Task<String> createTask() {  
        return new Task<String>() {  
            protected String call() throws Exception {  
                ...  
            }  
        };  
    }  
};
```

```
meaningService.start();
```

# Zaslanie úlohy do radu JAT

- Zaslanie úlohy z worker vlákna do JAT:
  - ▣ `Platform.runLater(Runnable úloha)`
- Úloha sa spustí, až keď sa vybavia úlohy pred tým

# Priebežné výsledky: Service aj Task

- Niektoré „observable“ premenné, ktoré vieme použiť ako model komponentov, resp. na nich odchytať udalosti zmeny aj vo vlákne GUI:
  - ▣ `ReadOnlyStringProperty messageProperty()`
    - meníme cez `updateMessage(String message)`
  - ▣ `ReadOnlyDoubleProperty progressProperty()`
  - ▣ `ReadOnlyDoubleProperty totalWorkProperty()`
  - ▣ `ReadOnlyDoubleProperty workDoneProperty()`
    - všetky 3 meníme cez `updateProgress(double workDone, double max)`
  - ▣ `ReadOnlyStringProperty titleProperty()`
    - meníme cez `updateTitle(String title)`
  - ▣ `ReadOnlyObjectProperty<V> valueProperty()`
    - meníme cez **`updateValue(V value)`**
  - ▣ `ReadOnlyObjectProperty<Worker.State> stateProperty()`
  - ▣ `ReadOnlyObjectProperty<Throwable> exceptionProperty()`

# Zadanie

- Stiahnite si z GitHubu poslednú verziu:
  - <https://gitlab.science.upjs.sk/peter.gursky/kopr2019>
- nasledujúce zadanie:
  - cviko09.zadanie
- Zrealizujte kontrolu gramatiky cez Service